UNIVERSITY OF OSLO
Department of Informatics

FLACOS'08 Workshop Proceedings

Research Report No. 377

Gordon J. Pace

Gerardo Schneider

ISBN 82-7368-337-0
ISSN 0806-3036

November 2008

Proceedings of

FLACOS'08

Second Workshop on Formal Languages and Analysis of Contract-Oriented Software

27–28 November 2008

Malta

Gordon J. Pace and Gerardo Schneider (editors)

Foreword

The 2nd Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'08) is held in Malta. The aim of the workshop is to bring together researchers and practitioners working on language-based solutions to contract-oriented software development.
The workshop is partially funded by the Nordunet3 project "COSoDIS" (Contract-Oriented Software Development for Internet Services) and it attracted 25 participants.

The program consists of 4 regular papers and 10 invited participant presentations. The regular papers were selected by the following programme committee:

Björn Bjurling	SICS, Sweden
Olaf Owe	University of Oslo, Norway (co-chair)
Anders P. Ravn	Aalborg University, Denmark
Gerardo Schneider	University of Oslo, Norway (co-chair)

Further information can be found at the workshop homepage: `http://www.ifi.uio.no/flacos08`.

Acknowledgments

We thank the Department of Computer Science, University of Malta for its support.

Welcome to Malta!
Gordon J. Pace and Gerardo Schneider

Table of Contents

Cc-Pi: A Constraint-Based Language for Contracts with Service Level Agreements*

Maria Grazia Buscemi[1] and Ugo Montanari[2]

[1] IMT Lucca Institute for Advanced Studies, Italy
m.buscemi@imtlucca.it

[2] Dipartimento di Informatica, University of Pisa, Italy
ugo@di.unipi.it

Abstract. Service Level Agreements are a key issue in Service Oriented Computing. SLA contracts specify client requirements and service guarantees, with emphasis on Quality of Service (cost, performance, availability, etc.). We overview a simple model of contracts for QoS and SLAs that combines two basic programming paradigms: name-passing calculi and concurrent constraint programming. In the resulting calculus, called cc-pi calculus, SLA requirements are constraints that can be generated either by a single party or by the synchronisation of two agents. We rely on a system of named constraints that equip classical constraints with a suitable algebraic structure providing a richer mechanism of constraint combination. Besides small examples, cc-pi has been applied to a Telco case study. The model allows to specify, negotiate, and enforce policies in complex scenarios where policy negotiations and validations can be arbitrarily nested.

1 Introduction

A key feature of the service oriented computing paradigm is the possibility of selecting and invoking services. Beside functional properties, services may expose non-functional properties including Quality of Service (QoS), cost, and security. Non-functional parameters play an important role in service discovery and binding. Indeed, a service requester might have minimal QoS requirements below which a service is not considered useful. Moreover, multiple services that meet the functional requirements of a requester can still be differentiated according to their non-functional properties. Service Level Agreements (SLAs) capture the mutual responsibilities of service provider and service requester with respect to non-functional properties, with emphasis on QoS values.

The terms and conditions appearing in a SLA contract can be negotiated among the contracting parties prior to service execution. In the simplest case, one of the two parties exposes a contract template that the other party can fill in with values in a given range. However, in general the two parties may need a real negotiation in which they pose arbitrary complex *policies*, namely SLA

* Research supported by the EU IST-FP6 16004 Integrated Project Sensoria

requirements and guarantees. If the parties fail to reach an agreement, they may weaken their policies. Moreover, during the service execution, the service usage is checked for compliance with the SLA defined at subscription time.

The *cc-pi calculus* [6, 7] is a model of contracts with QoS and SLAs that is also suited to study mechanisms for resource allocation. This model is inspired by two basic programming paradigms: name-passing calculi and concurrent constraint programming (`cc` programming) [11]. While the informal concept of constraint is widely used in a variety of different fields, a very general, formal notion of constraint system has been introduced in the `cc` programming paradigm. Basically, `cc` programming is a simple and powerful computing model based on a shared store of constraints that provides partial information about possible values that variables can take. Concurrent agents can act on this store by performing either a `tell` action (for adding a constraint, if the resulting store is *consistent*) or an `ask` action (for checking if a constraint is *entailed* by the store). As computation proceeds, more and more information are accumulated, thus the store is *monotonically refined*.

The cc-pi calculus enriches classical `cc` programming with a channel-based communication mechanism and a restriction operation à la pi-calculus [10] along with a possibly non-monotonic evolution of the constraint store. Specifically, cc-pi features a symmetric, synchronous mechanism of interaction between senders and receivers, where the sent name is 'fused' (i.e. identified) to the received name, and such an *explicit fusion* allows using interchangeably the two names. The entities involved in a SLA negotiation are modelled as communicating cc-pi processes and SLA guarantees and requirements are expressed as constraints that can be generated either by a single process or as a result of the synchronisation of two processes. Moreover, the restriction operator of the cc-pi calculus can limit the scope of names thus allowing for local stores of constraints, which may become global after a synchronisation.

The constraint systems adopted in cc-pi rely on *named c-semirings*. A c-semiring [2] is a commutative semiring with top element and such that the sum operation $\oplus$ is idempotent. Intuitively, the preference level associated to each variable instantiation is modelled as a value of the c-semiring; the combination of constraints is expressed by the c-semiring product $\otimes$, while the semiring sum $a \oplus b$ chooses the worst constraint better than a and b. Named c-semirings enrich classical c-semirings with a notion of *support* to express the relevant names of a constraint. Semiring-based structures can specify networks of constraints for defining constraint satisfaction problems and model fuzzy or probabilistic values, as well as Herbrand unifications.

A recent trend in Telecommunication is to adopt service-oriented technologies to expose capabilities (e.g. call control, sending/receiving messages, or access information on end users), implemented in a Telco network, to applications deployed in third party administrative domains. In such a context, network operators and third parties have to define SLAs in order to monitor the access and usage of Telco capabilities. We have applied the cc-pi calculus for specifying, negotiating, and enforcing contracts for Telco services.

Related work. Bistarelli and Santini [4] have presented a constraint-based model for SLAs as an extension of *soft* `cc` programming [3]. The proposed model includes operations quite different from those of the cc-pi calculus, such as those for relaxing the constraints involving a given set of variables and then adding a new constraint, and for checking if a constraint is not entailed by the store. Coppo and Dezani-Ciancaglini [8] have proposed a calculus of contracts by combining the basic primitives of the cc-pi calculus with the notion of sessions and session types to design communicaion protocols which assure safe and reliable communication sequences. Bacciu *et al.* [1] have developed a formalism for specifying the service guarantees and requester requirements on QoS and the negotiation mechanism. Unlike our model, their approach relies on fuzzy sets rather than on c-semirings. SLAng [12] and WSLA [9] are XML-based languages for defining SLAs at a lower level of abstractions. The elements of SLAng are also constraints on the behaviour of associated services and service clients, but their are specified in OCL. WSLA provides the ability to create new SLAs as functions over existing metrics. This is useful to formalise requirements that are expressed in terms of multiple QoS parameters. The semantics for expressions over metrics is not formally defined, though.

2 The CC-Pi Calculus

In this section we outline the main features of the cc-pi calculus. The interested reader can refer to [6] for a detailed presentation of the calculus.

The cc-pi calculus combines synchronous channel-based communication with primitives like `tell` and `ask` that are inspired by the constraint-based computing paradigms and that account for placing constraints and making logical checks. In more detail, a single cc-pi process `tell` $c.Q$ can place a constraint c (which corresponds to a certain requirement/guarantee) if c is consistent with the actual store and then evolve to process Q. The process `check` $c.Q$ behaves like `tell` $c.Q$ except for the fact that c is not added. Similarly, the process `ask` $c.Q$ checks whether c is entailed by the actual store of constraints and, in the positive case, becomes Q. Alternatively, two processes $P = \overline{p}\langle\tilde{x}\rangle.P'$ and $Q = p\,\tilde{y}.Q'$ that are running in parallel $(P \mid Q)$ can synchronise with each other on the port p by performing the output action $\overline{p}\langle\tilde{x}\rangle$ and the input action $p\,\tilde{y}$, respectively, where $\tilde{x}$ and $\tilde{y}$ stand for sequences of names. Such a synchronisation creates a constraint induced by the identification of the communicated parameters $\tilde{x}$ and $\tilde{y}$, if the store of constraints obtained by adding this new constraint is consistent, otherwise the system has to wait that a process removes some constraint (action `retract` c). Finally, a process $(x)P$ declares a local name x that can become public as a result of a synchronisation.

Underlying constraint system. The cc-pi calculus is parametric with respect to *named constraints*, which are meant to model different SLA domains. Consequently, it is not necessary to develop ad hoc primitives for each different kind of SLA to be modelled. A named constraint c is an element of a *named c-semiring*, namely a c-semiring structure and equipped with a notion of *support*

supp(c) that specifies the set of "relevant" names of c, i.e. the names that are affected by c. The notation $c(x, y)$ indicates that supp(c) = $\{x, y\}$. Formally, a c-semiring $S = \langle A, \oplus, \otimes, 0, 1\rangle$ is a commutative semiring with top element and such that $\oplus$ is idempotent. C-semiring values express a preference level, while the combination of constraints is expressed by the product operation and the sum of two constraints a and b chooses the worst constraint better than a and b. The relation $\preceq$ over constraints is defined as $a \preceq b$ if $a \oplus b = b$. Intuitively, $a \preceq b$ expresses that a is more constrained than b, or, more interestingly, that a entails b. A set C of named constraints is *consistent* if the product of the elements of C is different for the bottom element 0.

Soft constraints. Named constraints are particularly suited to specify soft constraint satisfaction problems. The key idea underlying constraint satisfaction problems is to solve a problem by stating constraints representing requirements about the problem and, then, finding solutions satisfying all the constraints. Soft constraint satisfaction problems are meant to express preferences rather than strict requirements or to provide a not-so-bad solution when the problem is overconstrained. Formally, given a domain D of interpretation for a set of names $\mathcal{N}$ and a c-semiring $S = \langle A, \oplus, \otimes, 0, 1\rangle$, a *soft* constraint c can be represented as a function $c = (\mathcal{N} \to D) \to A$ associating to each variable assignment $\eta = \mathcal{N} \to D$ (i.e. instantiation of the variables occurring in it) a value in A, which can be interpreted e.g. as a set of preference values or costs. Soft constraints can be combined by means of the operators of S. For instance, the interpretation of the constraint $c = x \leq \mathsf{a} \otimes \mathsf{b} \leq y$, where x, y are names in $\mathcal{N}$, a, b are domain values in D, the underlying c-semiring is $S = \langle\{\mathsf{False}, \mathsf{True}\}, \vee, \wedge, \mathsf{False}, \mathsf{True}\rangle$, and $\leq$ has the usual meaning of "less than or equal" on numbers, is that c is the function $(\mathcal{N} \to D) \to \{\mathsf{False}, \mathsf{True}\}$, with the assignment η such that $c\eta = \mathsf{True}$ if $\eta(x) \leq \mathsf{a}$ and $\mathsf{b} \leq \eta(y)$, while $c\eta = \mathsf{False}$ otherwise.

3 A Telecommunication case study

In this section we analyse a case study borrowed from the Telecommunication area. We show how to apply the cc-pi calculus for specifying, negotiating, and enforcing policies for Telco services. We start by introducing a service scenario called CallBySms.

The CallBySms service allows a mobile phone user to activate a voice call by sending an SMS message to a specific service number. The SMS message must contain a nickname of the person the user wishes to call. The service is able to automatically find the number associated with the nickname and to set up a third party call between the user and the callee. In order to keep privacy, the service does not know actual phone numbers, but only opaque-id representing users. The service in turn uses two services, ThirdPartyCall and ShortMessaging, for specifying the operations respectively necessary to set-up and control calls and to receive/send short messages. Figure 1 depicts a possible service scenario in which John wishes to call Mary and he knows that Mary's nickname is "sunshine".

1. The Third Party application subscribes the services that are used by the CallBySms service and signs a SLA contract with the Network Operator;
2. The CallBySMS service is activated and the Third Party application receives a service number, e.g. 11111;
3. Mary sends an SMS "REGISTER sunshine" to the service number 11111;
4. The service associates "sunshine" to the opaque-id of Mary;
5. John sends an SMS "CALL sunshine" to the service number 11111;
6. The service retrieves the opaque-id associated to "sunshine" and set-up a call;
7. John's phone rings; John answers and gets the ringing tone;
8. Mary's phone rings; Mary answers;
9. John and Mary are connected.

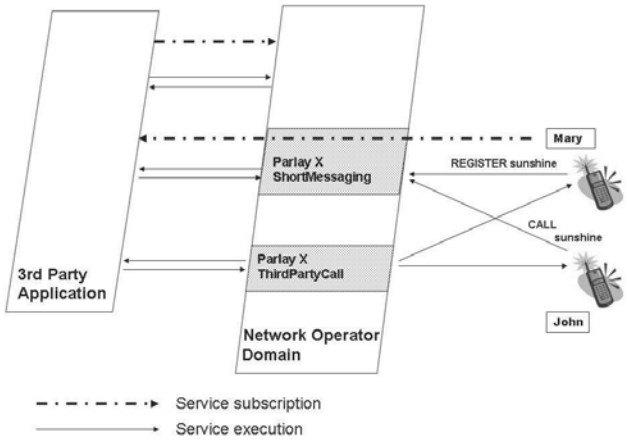


Fig. 1. CallBySms Service Scenario

Policies as constraints. We now focus on specifying and ensuring *time* policies. In [5] we address modelling and enforcement of other policies such as policies on *frequency*. For simplicity, hereafter we take the reference constraint system to be a classical constraint satisfaction problem by considering the named c-semiring of Boolean values. However, such constraint system can be easily generalised to soft constraint satisfaction problems by replacing the underlying c-semiring with an arbitrary c-semiring.

The constraint $c_{\text{time}}(i, f) = (7 \leq i \leq 9) \otimes (15 \leq f \leq 18)$ specifies the initial and final time ranges within which calls can be set up by end users. Similarly, $d_{\text{time}}(i, f) = (6 \leq i \leq 8) \otimes (17 \leq f \leq 19)$ states the time requirements of the third party. The result of combining these policies is the intersection of

the initial and final time ranges, which is expressed by the c-semiring product $e_{\text{time}}(i,f) = c_{\text{time}}(i,f) \otimes d_{\text{time}}(i,f) = (7 \leq i \leq 8) \otimes (17 \leq f \leq 18)$. Note that the constraint e_{time} is part of the SLA contract among the network operator and the third party application and it is validated by the operator domain once a call request from a end user is received. Other policies might depend on some network operator parameter while being related to the agreement of the third party with every end-user.

Cc-pi specification. We now show the main steps of the formalisation in cc-pi calculus of the policy negotiation and service execution scenario of CallBySms. We refer to [5] for a complete description of the specification.

The negotation phase between the third party application and the network operator consists of the two parties placing their own constraints and trying to synchronise on port x in order to export their local parameters. If the set of all such constraints induced by the synchronisation is consistent, the two parties have concluded a contract, which is expressed by the c-semiring product of all constraints:

$$\text{NO\_Neg}(x,z,t) = (i,f)\,(\texttt{tell}\ c_{\text{time}}(i,f)).\, x\langle i,f\rangle.\, \text{NO\_Acpt\_Reqst}(x,z,i,f,t)$$

$$\text{3rdPA\_Neg}(x) = (i',f')\,(\texttt{tell}\ d_{\text{time}}(i',f')).\overline{x}\langle i',f'\rangle.\, \text{3rdPA\_Acpt\_Reqst}(x)$$

The process $\mathit{Clock}_{\mathsf{T}}$ is meant to simulate the actual time by increasing of a time unit a variable t starting from its present value T. We assume this + operation automatically resets the clock by the end of the day:

$$\text{Clock}_{\mathsf{T}}(t) = \texttt{retract}\ (t = \mathsf{T}).\, \texttt{tell}\ (t = \mathsf{T}+1).\, \text{Clock}_{\mathsf{T}+1}(t)$$

After a service activation request by the third party application, the network operator is ready to accept registration requests from end-users and to forward them to the third party application. An end-user intending to register to CallBySms tries to synchronise with the network operator on z with its private identity *mary* and nickname *sunshine* as parameters. The network operator forwards this request to the third party application by sending on x the nickname and a private channel name *ch*, though not revealing the user's identity:

$$\text{Regist\_User}(z, \mathit{sunshine}) = (\mathit{mary})\,(\overline{z}\langle \mathit{mary}, \mathit{sunshine}\rangle.\, \text{Wait\_Calls}(\mathit{mary}))$$

$$\begin{aligned}\text{NO\_Acpt\_Reqst}(x,z,i,f,t) = (\mathit{id},\mathit{nn})(z\langle \mathit{id},\mathit{nn}\rangle.\overline{x}\langle \mathit{nn}\rangle.(&\\ &\text{NO\_Acpt\_Reqst}(x,z,i,f,t)\\ &\mid \text{NO\_Acpt\_Call}(i,f,t,\mathit{id},\mathit{nn})))\end{aligned}$$

$$\text{3rdPA\_Acpt\_Reqst}(x) = (\mathit{nn}')\,(x\langle \mathit{nn}'\rangle.(\text{3rdPA\_Acpt\_Reqst}(x))$$

A user who wants to call Mary but only knows her nickname is specified by a process sending its private name *john* on the public port *sunshine* and then waiting to be connected with *sunshine* on port *john*. The network operator verifies that the call request is within the legal time range. In case of success, the

network operator forwards the name *john* to the private port *mary* in order to connect the two users:

$$\begin{aligned}
\text{Wait_Calls}(mary) &= (cal')\,(mary\langle cal'\rangle.cal'\langle\rangle.\text{Wait_Calls}(mary)) \\
\text{Caller}(sunshine) &= (john)\overline{sunshine}\langle john\rangle.\overline{john}\langle\rangle.0 \\
\text{NO_Acpt_Call}(i,f,t,id,nn) &= (cal)\,(\texttt{check}\ (i \le t \le f).nn\langle cal\rangle.\overline{id}\langle cal\rangle. \\
&\qquad \text{NO_Acpt_Call}(i,f,t,id,nn))
\end{aligned}$$

The whole system S is given by the parallel composition of the two users, the clock and the processes specifying the policy negotiation followed by the processes modelling the service execution:

$$\begin{aligned}
S = (x,z,t,sunshine)\,&\text{Regist_User}(z,sunshine) \mid \text{Caller}(sunshine) \mid \\
&\texttt{tell}\ (t=0).\,\text{Clock}_0(t)) \mid \text{NO_Neg}(x,z,t) \mid \text{3rdPA_Neg}(x)
\end{aligned}$$

Note that our framework can be employed to model more complex negotiation scenarios, e.g. in which there is an arbitrary number of end-users or in which the third party application and the network operator may want to *retract* their initial policies and replace them with weaker constraints, in order to reach an agreement.

4 Conclusions

We have presented the cc-pi calculus, a constraint-based model of SLA contracts, and we have shown its flexibility by analysing a Telco case study. In [7], the cc-pi calculus has been equipped with an abstract semantics in the style of open pi-calculus, along with a symbolic transition system with contextual labels. We have also studied the expressiveness of the calculus: we have provided a reduction-preserving translation of `cc` programming [6] and a translation of the explicit fusion calculus by Gardner and Wischik which respects open bisimilarity [7].

We plan to further explore expressiveness issues by translating other calculi like the open pi-calculus and the applied pi-calculus and by proving that such translations preserve the behavioural equivalences defined for these calculi.

References

1. D. Bacciu, A. Botta, and H. Melgratti. A fuzzy approach for negotiating quality of services. In *Proc. TGC*, volume 4661 of *Lect. Notes in Comput. Sci.*, pages 200–217, 2007.
2. S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, 1997.
3. S. Bistarelli, U. Montanari, and F. Rossi. Soft concurrent constraint programming. *ACM Trans. Comput. Logic*, 7(3):563–589, 2006.
4. S. Bistarelli and F. Santini. A nonmonotonic soft concurrent constraint language for sla negotiation. In *Proc. CILC*, 2008.

5. M. Buscemi, L. Ferrari, C. Moiso, and U. Montanari. Constraint-based policy negotiation and enforcement for telco services. In *Proc. TASE*, pages 463–472. IEEE Computer Society, 2007.
6. M. G. Buscemi and U. Montanari. Cc-pi: A constraint-based language for specifying service level agreements. In *Proc. ESOP*, volume 4421 of *Lect. Notes in Comput. Sci.*, pages 18–32. Springer, 2007.
7. M. G. Buscemi and U. Montanari. Open bisimulation for the concurrent constraint pi-calculus. In *Proc. ESOP*, volume 4960 of *Lect. Notes in Comput. Sci.*, pages 254–268. Springer, 2008.
8. M. Coppo and M. Dezani-Ciancaglini. Structured communications with concurrent constraints. In *Post-Proc. of TGC*, 2008. To appear.
9. A. Keller and H. Ludwig. The WSLA framework: Specifying and monitoring service level agreements for web services. *Jour. Net. and Sys. Manag.*, 11(1):57–81, 2003.
10. R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77, 1992.
11. V. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. POPL*. ACM Press, 1990.
12. J. Skene, D. Lamanna, and W. Emmerich. Precise service level agreements. In *Proc. ICSE*, 2004.

A Tool for the Design and Verification of Composite Web Services

María Emilia Cambronero, Gregorio Díaz, Valentín Valero, and Enrique Martínez *

Departamento de Sistemas Informáticos
Escuela Politécnica Superior de Albacete
Universidad de Castilla-La Mancha
Campus Universitario s/n. 02071. Albacete, SPAIN
{MEmilia.Cambronero, Gregorio.Diaz, Valentin.Valero, Enrique.Martinez }@uclm.es

Abstract. This paper describes the main features of the Web Services Translation tool, WST for short, a tool for modeling and verification of Web Services systems with time restrictions. The different parts of WST are then presented, and a case study is used to show the potential of this tool. This case study is called "Dynamic Internet Purchase Site" and it allows us to see the capabilities of modelling, code generation and verification of the tool. This case study is a Web Service system with time restrictions, which are one of the main objetives of the WST verification part.

1 Introduction

Internet and Web technologies are a new way of doing business, more cheaply and efficiently, as enterprises can provide new and dynamic services in a faster way by the composition of Web Services. But B2B e-commerce is still emerging, and new software technologies are being required to support their development. Specifically, there is a need for effective and efficient means to abstract, compose, analyse, and evolve Web Services in an appropriate time-frame [8].

In this framework we can find some problems, which are the main motivation of the development of Web Services Translation tool:

- The interest in web services has grown in recent times as more and more intra/inter-organizational applications use this model, but little effort has been dedicated to systematically design and analyze web services systems.
- The analysis of Web Services Coordination and specifically, the timed restrictions that must be enforced in Web Services for which timed aspects are crucial for a correct functionality. This point can be covered by using the so called choreographies, which describe the composition of several existing Web Services in order to provide a Composite Web Service.
- The use of formal techniques bring rigour and consistency to system specification and implementation. Web services systems can also be described, analysed and implemented by using formal techniques. They allow us to have unambiguous Web services descriptions, which can be later checked for detecting errors or can be used to prove that some properties of interest hold.

Thus, the main goal of this work is to present a tool based on a model-driven methodology, which allows us to deal with these problems. WST tool covers different methodological phases for the design and implementation of Composite Web Services, following the software life cycle: design and implementation of choreographies with timed restrictions, and their validation and verification. In the design phase we use the Unified Modeling Language (UML 2.0 [13]), in order to model the system conforming to the initial analysis requirements in a proper way. Thus, WST supports the modelling phase by means of a UML sequence diagram editor.

* Supported by the Spanish government (cofinanced by FEDER founds) with the project TIN2006-15578-C02-02, and the JCCLM regional project PAC06-0008-6995.

The paper is structured as follows. A discussion of related work is shown in Section 2. In Section 3 the Web Services Translation tool is presented. Section 4 explains the application of WST tool. Finally, the conclusions and the future work are presented in Section 5.

2 Related Work

In the market we can find different Web Services tools. For instance, H. Foster et al. [5] show the design and implementation of a tool, called WS-Engineer, for a model-based approach to verifying compositions of web service implementations. The tool supports verification of properties created from design specifications and implementation models to confirm expected results from the viewpoints of both the designer and implementer.

In [6] Xiang Fu et al. present a tool, called WSAT, for analyzing and verifying composite web service specifications by using model checking techniques. In this case the specifications are written in WS-BPEL, and they are translated to Guarded Finite State Automata (GFSA), and the model checker SPIN [10] is then used to analyze and verify the system.

Another tool for the analysis of Web services systems is EA4B [7], which defines an execution log for WS-BPEL. The execution log can then be read for post-execution debugging or for near real-time monitoring. This tool can be integrated with static analysis tools such as WSAT, so error traces generated by WSAT can be translated to log files and visually displayed.

There is another tool, called WS-VERIFY [9], intended for the analysis of WS-BPEL specifications by using the NuSMV model checker [3]. The specifications written in WS-BPEL are translated into a formalism called WSTTS (Web Services Timed State Transition Systems), which are similar to timed automata.

In [1] a declarative service flow language (*DecSerFlow*) is also presented, which is used for monitoring purposes. In that work process mining techniques are used to check the conformance of service flows by comparing the specification written in *DecSerFlow* with reality. A tool supporting this language (*Declare*) is presented in [12].

3 Web Service Translation tool

Web Services Translation tool (WST) is an integrated environment for the modelling and verification of real-time systems. It allows us to model systems by using UML 2.0 sequence diagrams, then we can translate these diagrams into WS-CDL specification documents and, in turn, the WS-CDL specifications are translated into Timed Automata, which are then used to simulate and verify the system behaviour. This tool is available at http://www.dsi.uclm.es/retics/WST/.

In the generation of Web Services, as in the generation of any software system it is necessary to apply a methodology covering every phase of the life cycle. Figure 1 depicts a diagram that shows a schematic view of the top-down methodology implemented by WST, which consists of the following phases:

1. Analysis phase: In the analysis phase we use a technology based on goal models performing *requirement engineering*, KAOS [11] in order to capture the requirements that the system must fulfill.
2. Design phase: In the design phase we use the Unified Modeling Language (UML 2.0 [13]), and specifically sequence diagrams including some of their new capabilities, as the possibility of nesting frames to model the time restrictions of a system as well as the UML Profile for Schedulability, Performance, and Time (RT-UML [14]) in order to capture the time analysis requirements of systems in a proper way.
3. Choreography Implementation phase: we automatically translate the UML sequence diagrams into WS-CDL documents. The WS-CDL specification consist of the different parties plus its relationships and control structures used for structuring the communication process.

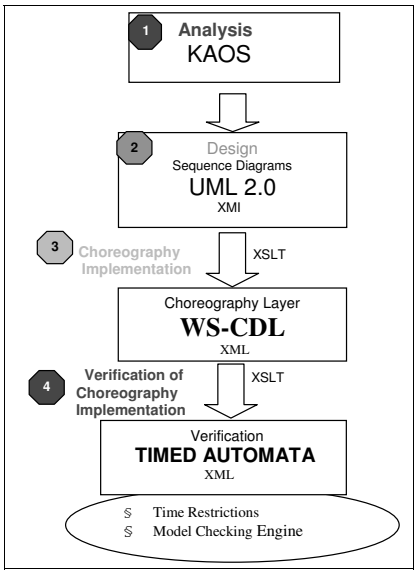


Fig. 1. WST-Methodology

Object	$\longrightarrow$ Role
Message	$\longrightarrow$ Relationship Type & Channel$^+$ & Interaction
Label and Time Constraint	$\longrightarrow$ Time Variable & Information Type & Expression
Frame "alt"	$\longrightarrow$ Choice
Frame "opt"	$\longrightarrow$ Workunit (without repeat condition)
Frame "loop"	$\longrightarrow$ Workunit (with a repeat condition)
Frame "par"	$\longrightarrow$ Parallel
Where the symbols +, | are BNF notation, and & is used to join information	

Table 1. UML 2.0 to WS-CDL Mapping rules

4. Verification of Choreography Implementation phase: the generated WS-CDL documents are translated into Timed Automata, which are used to simulate and verify some properties of interest by using the UPPAAL tool. The properties to check are those that we have established in the first step. If we detect some failures here, we return to the second step.

WST applies several XSL Stylesheets to an initial XML document to obtain another XML document. Let us now describe briefly the two main translations supported by the WST tool.

3.1 Obtaining WS-CDL documents from UML 2.0 sequence diagrams

WST translates a Web Service description with timed restrictions written by using UML sequence diagrams into a more commonly used language for Web Services choreographies description, WS-CDL. The WST tool uses three XSL Stylesheets in cascade in order to obtain the WS-CDL document from the XMI UML document. The UML file is structured in different sections, from which we obtain the elements that compose the WS-CDL document. Table 1 contains a scheme illustrating how the main elements of UML are translated into WS-CDL (for more details see [2]).

3.2 Obtaining Timed Automata from WS-CDL specifications

WST also translates the WS-CDL specifications into formal descriptions (Timed Automata) supported by the UPPAAL tool. In order to obtain these formal descriptions, WST uses XSL Stylesheets in cascade that are applied over the WS-CDL specifications. These specifications contain different elements like role types, channels, variables, control structures, etc, which are translated into the different elements of Timed Automata: templates, channels, variables, states, transitions, guards, and so on. A complete description of this translation can be found in [4].

4 Case Study: Dynamic Internet Purchase Site

Internet sites have been used to provide several functionalities. Among them, the most typical are searchers, personal pages (blogspots), information pages, government pages and pages for selling products. This last kind of sites has generated a great expectancy over financial markets due to the chance that represents to cover different national markets by using the Internet. This idea has been one of the most important reason to achieve what we call nowadays the "globalization" phenomenon. This is the reason to choose selling sites as a case study. This scenario is based on typical selling sites as "Amazon.com". The main features that we can discover in these sites are product search, cart, payment gateway, checkout and carrier facilities. Furthermore, we have modified it with a new feature that makes our site dynamic. With this feature, we can modify our site easily by adding or removing new parties to it. This site accepts three kind of parties, each of them playing a different role: provider, gateway or carrier.

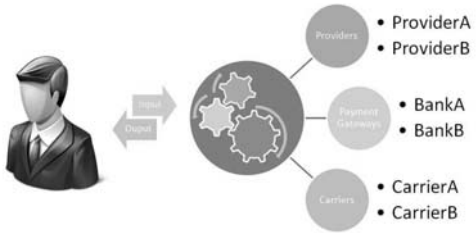


Fig. 2. Relational Diagram for a Selling Site.

Providers supply the site with the products to be sold and the information about them: price, selling price, description, number of days to be supplied, expiry date, etc.

Web Service	Parameters	Output	Description
Attach	Type and Info	ID	It allows to attach a new party by using the type (Provider, Payment Gateway or Carrier) and the info parameters. As a result this service returns the identification.
Detach	Type and ID	none	It detaches the party passed as an argument.
Status	none	Info	It shows the state of the selling site: Online, Off-line or Maintenance.
Info	Type and ID	Info	It returns as a result the info of a certain party identified by the parameters.
Request_PI()	none	List	It elaborates a list of the products and returns it.
AddItem	ID and Amount	none	It adds a product to the user cart.
RemoveItem	ID and Amount	none	It removes a product from the user cart.
PaymentAck	PayID	none	For informing the selling site if a payment has been successfully.
Delivered	ID	none	The carrier informs that the product identified by the identification has been delivered.

Table 2. Selling Site Web Services

Gateways let user introduce the personal and confidential information in order to perform the payment.

Carriers transport the product to users within the time bound established with the Internet site. To accomplish this task, the Internet site requests the time of delivery before asking the carrier to deliver the product. If the carrier fails to deliver the product in this interval, then the site cancels the delivery and orders the refund to the user.

Web Service	Parameters	Output	Description
ProductsInfo	none	List	It returns a list with the products provided by the provider.
Request	Id and Amount	none	It evaluates the request and establishes whether the request can be immediately fulfilled or not. In case of a negative response, it returns the number of days to provide the request.
Booking	ID and Amount	boolean	This service ask for selling a certain product and amount. If an error occurs, then a negative number is produced.
Status	none	Info	It shows the state of the provider: Online, Off-line or Maintenance.

Table 3. Providers Web Services

Figure 2 illustrates the relationships among the different roles and parties. The Internet site runs as a central system where all actions must be coordinated to achieve the common goal. The user provides inputs to this system and gets outputs as a result. The Internet site contacts providers for products, payment gateways for checkout process and carriers for offering users the carriers facilities. Tables 2, 3, 4 and 5 summarize the different services offered by each party.

Web Service	Parameters	Output	Description
Request	UsrInf and Total	PayID	It contacts with the bank and accesses to the user identification with the confidential information of the user. If the purchase is permitted by the bank, then the transaction is performed and a payment identification returned.
Status	none	Info	It shows the state of the payment gateway: Online, Off-line or Maintenance.

Table 4. Payment Gateway Web Services

Web Service	Parameters	Output	Description
Request	UsrInf and ProdInf	Days	It returns the number of days to deliver the product. If the address is not reachable by the carrier a negative number is produced.
Delivering	UsrInf and ProdInf	DelvID	It is a delivering order that produces a Delivering Id.
Delivering_Stt	DelvID	DelvInfo	It returns the location of the package.
Status	none	Info	It shows the state of the carrier site: Online, Off-line or Maintenance.

Table 5. Carrier Web Services

A sequence diagram has been elaborated that models the choreographies that could be generated. In this sense, Figure 3 depicts a scenario where a user performs a typical purchase in the selling site. The user starts by requesting the product list, then he selects a product and adds it to his cart. Once the cart has at least one product, the user can remove the product from the cart, adds a new product or performs the checkout. If this final option is performed, then the scenario starts the payment procedure by contacting the payment gateway with the total amount and the seller information. Then, the user supplies the credential to the payment gateway and an acknowledgement is sent to the selling site. If the payment has finished successfully, then a carrier is requested for delivering the product within 48 hours. Within this period, once the carrier has delivered the package, he informs to the selling site about it and the process finishes. But, if the period has expired, then the seller refunds the money to the user and informs the carrier to abort the delivery. Other two possible cases captured by this scenario are: first, the possibility of a negative acknowledgment from the payment gateway, in this case the purchase is aborted; second, the user remains idle too much time and the session expires.

The translation of this scenario into WS-CDL and Timed Automata is depicted in Figure 4. In the left-hand side of the figure, we can observe that the XMI definition of the sequence diagram is transformed into a WS-CDL specification. And in the right-hand side of the figure, the generated specification is subsequently translated into a timed automata specification supported by the UPPAAL tool.

An example of a simulation of this scenario is shown in Figure 5. This figure shows a snapshot of the Uppaal tool at the simulation tab where the scenario is running and several messages are being sent and received between the user and the selling site at that moment. At the left upper side of this figure we can see the automata running in parallel. These automata correspond to three of the parties involved in the scenario (InternetSite, User, and Carrier).

Another snapshot is shown in Figure 6, where we can see the verification process for different requirements that the scenario must fulfil in order to prove whether its functionality is correct or not. In this sense, this figure shows at the top part the formulas that are being verified and the results can be found at the bottom. For example, with this tool we can verify if the scenario can finish at correct or incorrect situations. These situations are represented in the figure by the three first formulas; the first

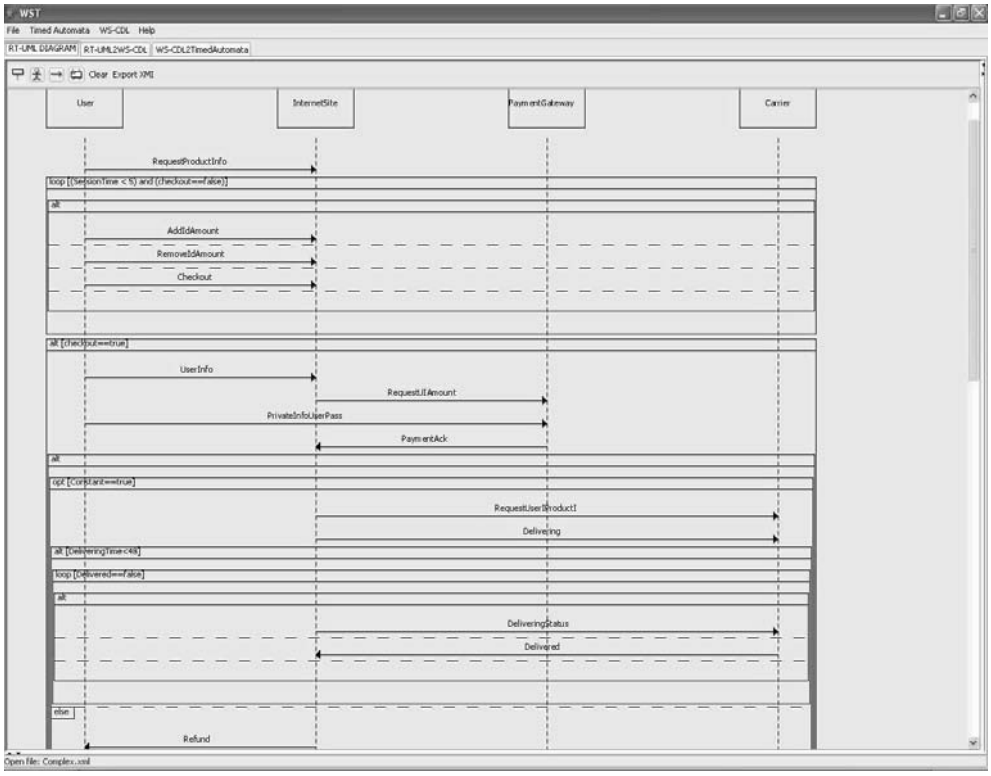


Fig. 3. A piece of the sequence diagram for the study case.

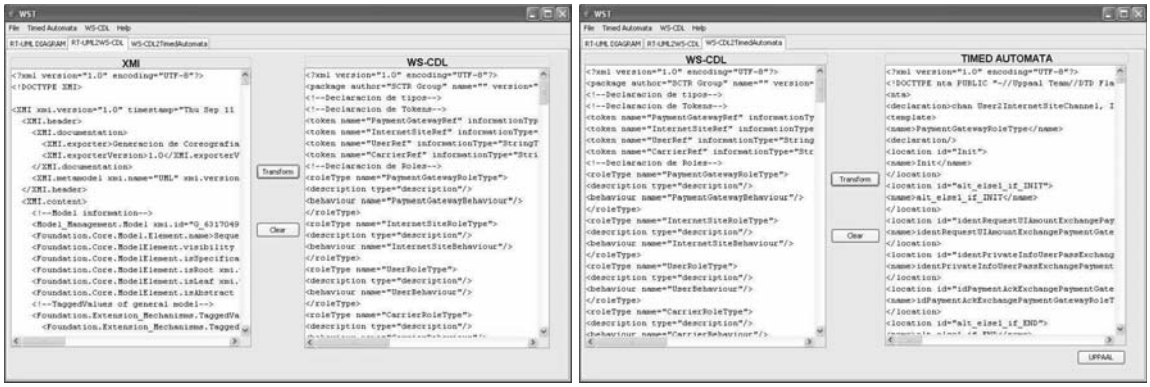


Fig. 4. Translation from XMI sequence diagram into WS-CDL and from WS-CDL into Timed Automata, respectively.

corresponds to the successfull delivery, the second to the session expiration and the third to the time exceeded for the delivery. The second and the third situation should be taken into account despite of being unsuccessfull scenarios.

We can see that during the verification we sometimes obtain negative responses to our requirements. This cases should be studied in detail by following the next three steps. The first step is to generate the counterexample. The second step consists of following the counterexample in order to detect where, when and how the error occurs. And last, the third step consists of deciding if it is a real error or an error occurring due to an inconsistency in the design of the checked requirement. In the case of a real error, it is necessary to modify the diagram and recheck the requirement to verify if the error has been fixed. For instance, in the scenario under study, we have found several errors and this errors allow the

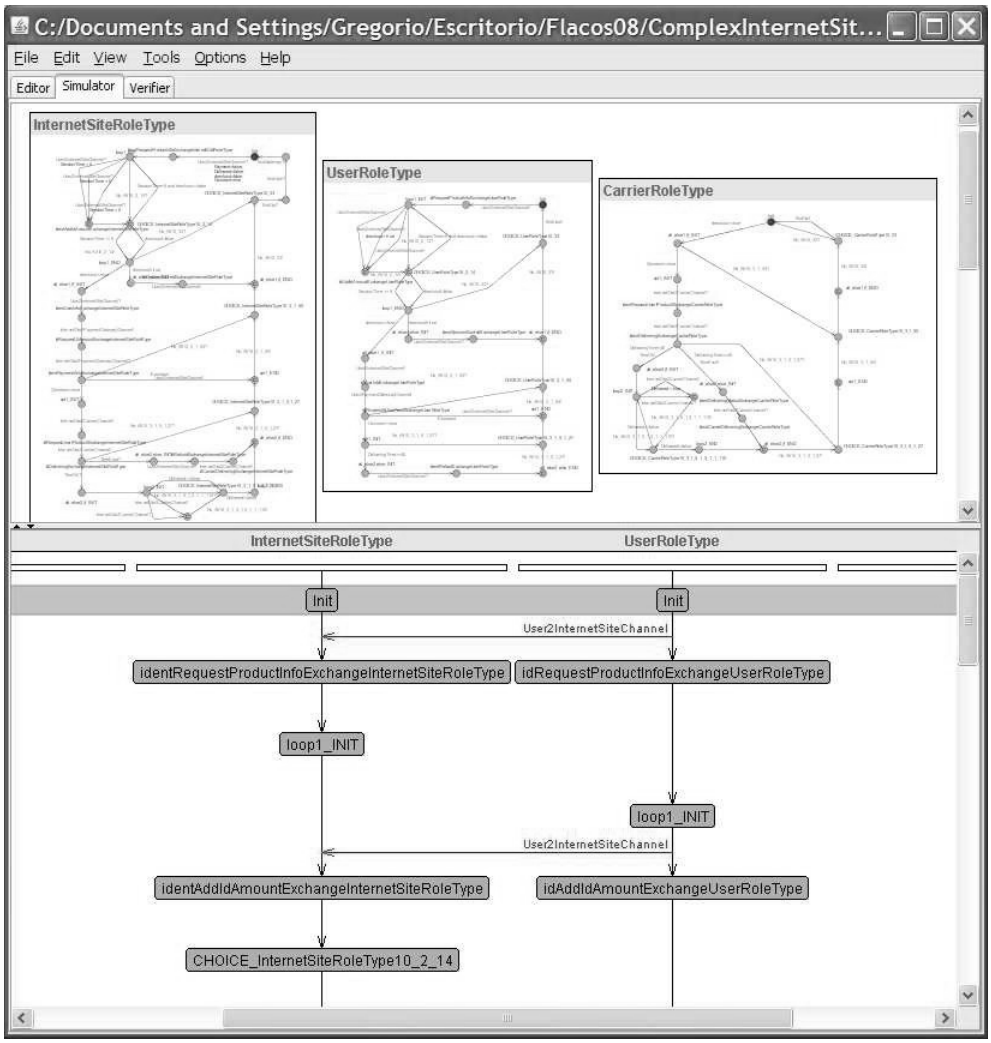


Fig. 5. Uppaal Simulation for the scenario and Uppaal Verification process for the scenario.

developers to detect the model inconsistencies in an early phase of the development, decreasing the number of errors in later phases.

5 Conclusions and Future Work

In this work, we have presented WST as a tool to support Web Services designs with time restrictions. The starting point in WST are UML 2.0 sequence diagrams, which allow us to model the communication among the parties, and also the control structures of the communication processes and the variables that capture the time and control conditions. After modelling the sequence diagrams, they can be translated into Web Services descriptions. These descriptions are a global view of the communication processes. Finally, the descriptions will be translated into a formal specification supported by a model checking engine. This engine allow developers to validate the design of the Web Services.

To show these features, we have used a real example based on an Internet selling site where we have designed a scenario. In this scenario, a user adds several products to a cart, performs the payment and waits the delivery of the product. We have introduced two time restrictions. the first based on the expiration of an Internet session and the second based on the maximum delay of the delivery.

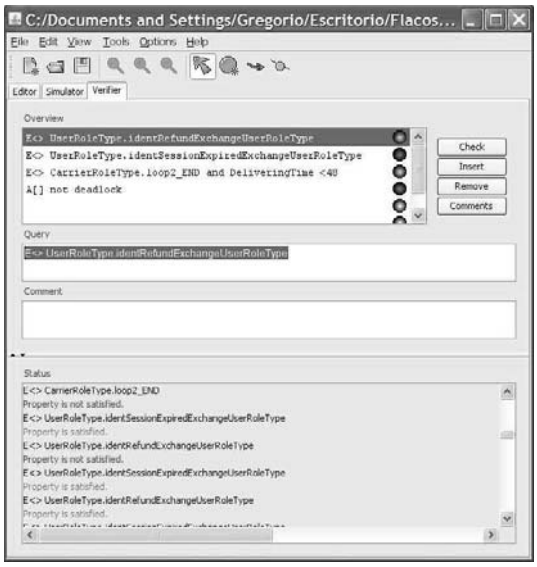


Fig. 6. Uppaal Verification process for the scenario.

References

1. W.M.P. van der Aalst and M. Pesic. *Specifying and Monitoring Service Flows: Making Web Services Process-Aware*. In the book *Test and Analysis of Web Services*, pp. 11–55. Editors: L. Baresi and Elisabetta Di Nitto. Springer. 2007.
2. M. E. Cambronero, G. Díaz, J. J. Pardo, V. Valero, and Fernando Pelayo. *RT-UML for Modeling Real-TimeWeb Services.* Proceedings of Modeling, Design, and Analysis for Serviceoriented Architecture Workshop, mda4soa, SCC 2006, pp. 131–139, Chicago, USA, 2006. IEEE Computer Society.
3. A. Cimatti, E.M. Clarke, F. Giunchiglia and M. Roveri. *NuSMV: A New Symbolic Model Checker*. International Journal on Software Tools for Technology Transfer (STTT), vol 2, no.4, pp. 410–425. 2000.
4. G. Díaz, J.J. Pardo, M.E. Cambronero, V. Valero, and F. Cuartero. Verification of Web Services with Timed Automata. In *ENTCS, vol: 157, issue: 2, pages 19–34*. 2005.
5. H. Foster, S. Uchitel, J. Magee, and J. Kramer. *WS-Engineer: A Tool for Model-Based Verification of Web Service Compositions and Choreography*. 29th IEEE/ACM International Conference on Software Engineering (ICSE), pp. 771–774. IEEE Computer Society/ACM Press, 2006.
6. X. Fu, T. Bultan, and J. Su. *Wsat: A tool for formal analysis of web services.* Proceedings of Computer-Aided Verification (CAV), vol. 3114 of Lecture Notes in Computer Science, pp. 510–514. Springer–Verlag, 2004.
7. A. Gravel, X. Fu, and J. Su. *An analysis tool for execution of bpel services.* IEEE Joint Conference on E-Commerce Technology (CEC) and Enterprise Computing, E-Commerce and E-Services (EEE), pp. 429–432, Tokyo, Japan, 2007. IEEE Computer Society.
8. R. Hamadi and B. Benatallah. *A Petri Net-based Model for Web Service Composition.* In Proceedings of the 14th Australasian database conference, vol. 17, pp. 191–200. 2003.
9. R. Kazhamiakin. *Formal Analysis of Web Service Compositions*. PhD. Dissertation, Univ. of Trento. 2007.
10. G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
11. A. van Lamsweerde A. Dardenne and Stephen Fickas. *Goal-directed requirements acquisition.* Selected Papers of the Sixth International Workshop on Software Specification and Design. Science of Computer Programming, vol. 20, Issue 1-2, pp. 3–50. 1993
12. M. Pesic, H. Schonenberg and W.M.P. van der Aalst. *DECLARE: Full Support for Loosely-Structured Processes*. Proc. of 11th International Enterprise Distributed Object Computing Conference (EDOC'07), page 287. IEEE Computer Society Press. 2007.
13. OMG. *UML 2.0 Superstructure proposal v.2.0.*, January 2003.
14. *UML Profile for Schedulability, Performance, and Time Specification, Version 1.1.* http://www.omg.org/docs/smsc/04-12-05.pdf.

Permission to Speak: An Access Control Logic*

Nikhil Dinesh Aravind Joshi Insup Lee
Oleg Sokolsky
Department of Computer and Information Science
University of Pennsylvania
{nikhild,joshi,lee,sokolsky}@seas.upenn.edu

October 26, 2008

Abstract

The modality of *saying* is central to access control logics. In this paper, we investigate the interaction of saying with the deontic modalities of *obligation* and *permission*. The motivation is to provide a unified formalism for phenomena that have been studied separately in the literature – (a) representation in access control, e.g., delegation and speaking for, (b) positive and negative permissions, and (c) conformance in the presence of iterated deontic modalities, e.g., "required to forbid". The central idea is to use statements that permit or require other statements. We propose two axiom schemas that transfer permissions and obligations from one set of laws to another. Policies are expressed in a non-monotonic formalism that accommodates reasoning about the transfer of permissions.

1 Introduction

Access control is an important problem in trust management systems. Informally, a trust management system involves a set of actors or principals, and a set of controlled or regulated actions, e.g., accessing medical information, or downloading a song. The goal of such a system is to administrate requests to perform actions. Trust management systems are commonly decomposed into two (interacting) components [1]: (a) *authentication* - determining the source of a request, and (b) *access control* - determining whether a request is permitted according to a policy. We focus on the problem of access control, which involves representing policies and evaluating requests. However, we will consider policies containing both *obligations* and *permissions*, rather than the usual access control setting where permission is the more important deontic modality.

The motivation for this work is to provide a unified formalism for phenomena that have been considered separately in the literature – (a) representation in access control [3, 1, 2, 6, 11, 13], e.g., delegation and speaking for, (b) positive and negative permissions (cf. [5]), and (c) conformance in the presence of iterated deontic modalities [16], e.g., "required to forbid". The central idea of this paper is that these phenomena involve the interaction between the modalities of *saying* and *permission*. We discuss each item in turn to illustrate the connection.

Representation in Access Control: While there are a wide variety of access control logics, one commonality that stands out is a notion of *saying* [1]. We can express the fact that a principal makes a statement. $\text{says}_{l(A)}\varphi$ denotes that principal A says φ in the set of laws $l(A)$.[1] B represents A on φ is expressed as $\text{says}_{l(B)}\varphi \Rightarrow \text{says}_{l(A)}\varphi$, where $\Rightarrow$ is the implication connective of the underlying logic. *Speaking for* is a case of representation where one principal represents another on all statements. In [1], B speaks for A is expressed using second-order quantification, i.e., $\forall\varphi : \text{says}_{l(B)}\varphi \Rightarrow \text{says}_{l(A)}\varphi$. While there are a few alternatives in formalizing *speaking for* [1, 3, 10], we will recast it in terms of the interaction between saying and permission. An advantage of this analysis is that we can relate problems in access control policies to problems with regulatory texts in general.

We derive *speaking for* by taking a different view on *representation*. Specifically, we add an axiom schema to a propositional modal logic, which allows us to express *speaking for* using a propositional formula. The axiom

*This research was supported in part by ONR MURI N00014-07-1-0907 NSF CCF-0429948 and ARO W911NF-05-1-0158.

[1] We relativize speaking to a set of laws rather than a principal, i.e., $\text{says}_{l(A)}\varphi$, rather than $\text{says}_A\varphi$. This lets us use saying to reason about specific statements [9], and avoid the algebra over principals in [3, 10].

that we propose involves the interaction between saying and permission. We say that B represents A on φ iff A says that B is *permitted* to say φ, i.e., $\text{says}_{l(A)}(\mathcal{P}_B(\text{says}_{l(B)}\varphi))$, where $\mathcal{P}_B\psi$ is read as B is permitted to bring about ψ. Our *the axiom of transfer* states that if A says that B is permitted to say φ, and B says φ, then A says φ, which we express in our logic as

$$(\text{says}_{l(A)}(\mathcal{P}_B(\text{says}_{l(B)}\varphi)) \wedge \text{says}_{l(B)}\varphi) \Rightarrow \text{says}_{l(A)}\varphi$$

The axiom of transfer is intended for a particular sense of speaking/saying, i.e., *speaking on someone's behalf*. This sense of saying is the usual one in access control. To simplify matters, we do not explicitly represent the principal on behalf of whom a statement is being made. *B speaks for A* is expressed as $\text{says}_{l(A)}(\mathcal{P}_B(\text{says}_{l(B)}\bot))$, i.e., A says that B is permitted to say anything ($\bot$). We motivate our approach by showing how we can express constructs that have been examined in the literature on deontic logic.

Positive and Negative Permission: The intuitive definition of permission as the dual of obligation, i.e., $\mathcal{P}_A\varphi = \neg\mathcal{O}_A\neg\varphi$ ($\mathcal{O}_A\varphi$ is read as "A is obligated to bring about φ") is known to be inadequate (cf. [5]). It works fine for explicitly given permissions. However, for implicit permissions further distinctions need to be made. The most common distinction is between positive and negative permission. Can we conclude that and action foo not mentioned in the law is permitted? In one sense (positive permission), the answer is "no", because no explicit permission has been given. In another sense (negative permission), the answer is "yes", because the principal has not been explicitly forbidden from performing foo.

In our approach, the two kinds of permission are distinguished by varying the scope of negation. To establish positive permission, we determine whether $\varphi \Rightarrow \text{says}_{l(H)}(\neg\mathcal{O}_A\neg\text{foo})$ provable?, while for negative permission, we would establish that $\varphi \Rightarrow \text{says}_{l(H)}(\mathcal{O}_A\neg\text{foo})$ *not* provable? In Section 2, we use the formalism of [9] to reason about provability and its negation. The negation of provability is needed to express *didn't say*. In other words, H didn't say φ iff $\text{says}_{l(H)}\varphi$ is not provable from H's statements. The discussion in [5, 15] suggests to us that the relationship between negative permission and *didn't say* is known, but to our knowledge, an explicit representation of saying has not been carried out in a deontic logic.

Iterated Deontic Modalities: Marcus [16] pointed out a problem in formalizing iterated deontic modalities. We relate it to the distinction between the two senses of permission. Consider the following statement: *A should not allow her child (B) to play near the road*. Which sense of permission is appropriate here? Using positive permission, we get "A should not explicitly permit B to play", which is inadequate. Negative permission is appropriate here – "A should not *not require B not to play*", i.e., "A should require B not to play". To accommodate such reasoning, the two kinds of permission need to be distinguished in the syntax of the logic. We note that [16] argues for a distinction between senses of obligation rather than permission. In our approach, the various senses are distinguished by varying the scope of negation.

We now discuss the relationship between representation and iterated permissions. Suppose *a hospital (H) permits a patient (A) to permit her mother (B) to access her information*. We will rephrase the permission as follows: H says that A is *permitted to say* that B is permitted to access her information. Formally, this is represented as $\text{says}_{l(H)}(\mathcal{P}_A(\text{says}_{l(A)}(\mathcal{P}_B\text{access})))$. If A does indeed permit access to her mother $(\text{says}_{l(A)}(\mathcal{P}_B\text{access}))$, we will conclude $\text{says}_{l(H)}(\mathcal{P}_B\text{access})$ using the axiom of transfer, i.e., H permits access to B. As a result, iterated permissions are related to representation, i.e., "H permits A to permit B to do φ" iff "A represents H in permitting B to do φ".

Outline: In Section 2.1, we present the axiomatization of the modal operators for saying and obligation. We introduce two additional axioms that describe the interaction between the two modalities. In Section 2.2, we integrate the axiomatization into a logic programming approach of [9], to describe the process of saying. Then, in Section 3, we discuss our formalism in the context of related work. We consider two examples from existing access control logics and their representation in our formalism. We also consider the treatment of iterated deontic modalities [16] and argue that the partial solution provided in this paper is sufficient for access control applications.

2 A Logic for Access Control

In this section, we develop an access control logic, in the form of two interacting components – (a) the inference component, which involves the choice of appropriate axioms, and (b) the saying component, which is used to represent policies. Figure 1 shows the interaction between the components of the access control system. There are two kinds of actions of interest – (1) operational acts, e.g., downloading a song, and (2) speech acts. The

operational acts are described using a state, which contains the interpretation of predicates, and the speech acts are described using laws.

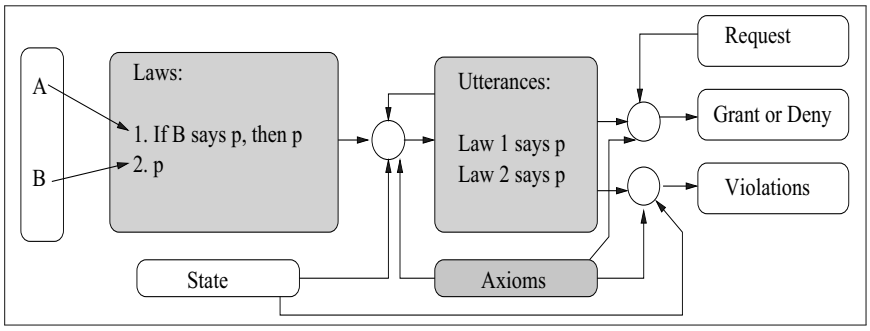


Figure 1: Interaction between the components of the access control system

A principal speaks by introducing laws. In Figure 1, the principals A and B introduce the laws 1 and 2 respectively. The laws are evaluated using the axioms to produce a set of *utterances*, i.e., what the laws say. To determine what a principal says, we look at what her laws say, e.g., B says p iff "Law 2 says p" is provable from the utterances using the axioms. A set of laws can be thought of as a logic program, and utterances as the extensions that result from the program (via a fixed point computation). Once we have the utterances, there are several decision problems of interest. *The access control problem* is to decide whether a request is permitted by the set of utterances. *The conformance problem* is to decide whether operational and speech acts satisfy the obligations imposed by the utterances, and if they do not, violations are reported.

2.1 The Inference Component – Syntax and Axioms

In this section, we develop a predicate logic with two modalities *saying* and *obligation*. We allow formulas with free variables, but no quantifier over objects. The quantification over objects is carried out in the process of saying (Section 2.2), which uses provability in the propositional subset of the language defined here. We begin by defining the syntax:

Definition 1 (Syntax) *Given sets $\Phi_1, ..., \Phi_n$ (of predicate names), countable sets of variables X, object names O, boolean variables B, a finite set of identifiers ID, and a function $l : O \to 2^{Id}$, the language $L(\Phi_1, ..., \Phi_n, X, O, B, l, ID)$, abbreviated as L, is defined as follows:*

$$\varphi ::= p(y_1, \ldots, y_j) \mid b \mid \varphi \wedge \varphi \mid \neg\varphi \mid \text{says}_{Id}\, \psi \mid \text{says}_{l(y)}\, \psi$$
$$\psi ::= \varphi \mid \psi \wedge \psi \mid \neg\psi \mid \mathcal{O}_y \varphi$$

where, $p \in \Phi_j$, $(y_1, ..., y_j) \in (X \cup O)^j$, $y \in X \cup O$, and $b \in B$. We assume that $X \cap O = \emptyset$. The set of formulas obtained from each BNF rule are referred to as L_φ and L_ψ respectively, and $L = L_\varphi \cup L_\psi$.

Disjunction $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$ and implication $\varphi \Rightarrow \psi = \neg\varphi \vee \psi$ are derived connectives. $\mathcal{O}_y\varphi$ is read as "φ is obligated of y". Permission is defined as the dual of obligation, i.e., $\mathcal{P}_y\varphi = \neg\mathcal{O}_y\neg\varphi$. The saying operators are understood as follows. Principals speak by introducing identified laws (Section 2.2), thus $\text{says}_{Id}\,\varphi$ is read as "φ is said in the laws Id". $l(A)$ is the set of laws introduced by the principal $A \in O$, allowing us to express "A says φ".

Note that the BNF rules ensure the alternation of *obligation* and *saying* modalities, e.g., $\mathcal{O}_y\, \text{says}_{l(y)}\, \mathcal{O}_z\varphi \in L$, but $\mathcal{O}_y\mathcal{O}_z\varphi \notin L$. Following von Wright [19], we understand obligations as applying to actions and their consequences. The language L_φ (obtained from the first BNF rule) describes actions – (a) atomic actions, (b) combinations of actions (using connectives), or (c) *saying*, which is (a consequence of) a speech act. An obligation is an opinion, which is created via a speech act, but is not an act by itself.

The statements in L will be used in *the inference component* of access control, i.e., to determine what has been said. In other words, we will be given a set of utterances U and a question ψ, and we need to determine whether

$U \Rightarrow \psi$ is *provable*. We focus on provability for the propositional subset of L, i.e., without variables and function applications. The propositional subset of L has the modalities $\text{says}_{Id}\,\varphi$ (for all $Id \subseteq ID$), and $\mathcal{O}_A(\varphi)$ (for all $A \in O$).

A1 All substitution instances of propositional tautologies.

A2 $\mathcal{Q}(\varphi \Rightarrow \psi) \Rightarrow (\mathcal{Q}(\varphi) \Rightarrow \mathcal{Q}(\psi))$ (for all modalities $\mathcal{Q}$)

A3 $\text{says}_{Id}\,\varphi \Rightarrow \text{says}_{Id'}\,\varphi$ (for all $Id \subseteq Id'$)

A4 $\mathcal{O}_A\varphi \Rightarrow \mathcal{P}_A\varphi$ (for all $A \in O$)

A5 $(\text{says}_{Id_A}(\mathcal{P}_B\,\text{says}_{Id_B}\,\varphi) \wedge \text{says}_{Id_B}\,\varphi) \Rightarrow \text{says}_{Id_A}\,\varphi$ (for all $\{A, B\} \subseteq O$, $Id_A \subseteq l(A)$, and $Id_B \subseteq l(B)$)

A6 $\text{says}_{Id_A}(\mathcal{P}_B\,\text{says}_{Id_A}\,\varphi) \Rightarrow \text{says}_{Id_A}\,\varphi$ (for all $\{A, B\} \subseteq O$, and $Id_A \subseteq l(A)$)

R1 From $\vdash \varphi \Rightarrow \psi$ and $\vdash \varphi$, infer $\vdash \psi$

R2 From $\vdash \varphi$, infer $\vdash \mathcal{Q}(\varphi)$ (for all modalities $\mathcal{Q}$)

Figure 2: Axiomatization of the propositional fragment of L.

We adopt the axiomatization in Figure 2. **A1** and **R1** give us propositional reasoning. **A2** and **R2** are common to both saying and obligation. **A3** and **A4** are specific to saying and obligation respectively. Finally, **A5** and **A6** describe the interaction between the two modalities. We will now discuss the axioms in the context of related work.

Axioms for Saying: The axioms **A1** and **A2**, together with the rules **R1** and **R2**, gives us the modal logic **K**. The **K** axiomatization was used by Abadi et al [3] as a basis for all (classical) access control logics. Further motivation comes from our previous work [9]. In [9] and in Section 2.2 here, to describe policies, we evaluate $\text{says}_{Id}\,\varphi$ using provability. Given a set of formulas U, $\text{says}_{Id}\,\varphi$ is true w.r.t. U iff $\bigwedge U \Rightarrow \text{says}_{Id}\,\varphi$ is provable. The **K** axiomatization is sound w.r.t. this definition. **A3** says that if φ is said by the statements (Id), then φ also holds according to a larger set of statements (Id'). This axiom is also sound w.r.t. [9].

Obligation and Its Interaction with Saying: The **K** axiomatization, together with **A4**, gives us the the modal logic **KD**. This axiomatization is common to many systems, giving it the name Standard Deontic Logic (c.f. [12]).

The main focus of this work is on the problem of *iterated deontic modalities* [16]. Our goal is to provide a partial solution, as needed for access control. We characterize the interaction between *saying* and *obligation* with two axioms. *The transfer axiom*, **A5**, is read as "If A says that B is permitted to say φ, and B says φ, then A says φ". As we discussed in Section 1, **A5** is needed to accommodate *speaking for* and *delegation*, and we will discuss examples in Section 3.1. *The self-respecting axiom*, **A6**, is read as "If A permits B to say φ using A's laws, then A says φ". **A6** ensures that statements in $l(B)$ do not (inadvertently) interfere with the consequences of statements in $l(A)$.

Provability: The process of saying (Section 2.2) relies on provability in the language L. We say that φ is provable (denoted $\vdash \varphi$), if φ is an instance of the axioms **A1**-**A6** or follows from the axioms using the rules **R1** and **R2**.

In the full paper, we establish the decidability of the provability question; that is, given $\varphi \in L$ which is propositional, $\vdash \varphi$ is decidable. There, we provide a Kripke semantics for which the axiomatization is sound and complete. As in [10], semantics is used to show that a statement is not provable. The full paper also provides the discussion of the computational complexity of testing satisfiability of φ.

2.2 The Saying Component - Policies and Conformance

In this section, we briefly discuss the representation and evaluation of policies or regulations. The result of evaluating regulation is a set of annotations or utterances, from which we can determine conformance. The formalism developed here is an extension of [9], and is a generalized form of logic programming. Logic programs are popular in representing regulatory texts [18, 17, 12], and access control policies [13, 7, 4].

Definition 2 (Syntax of Regulation) *Given a finite set of identifiers ID, a body of regulation Reg is a set of statements such that for each $\mathrm{id} \in ID$, there exist $\varphi \in L_\varphi$ and $\psi \in L_\psi$ such that:* $\mathrm{id} : \varphi \rightsquigarrow \psi \in Reg$

$\mathrm{id} : \varphi \rightsquigarrow \psi$ is read as: "the precondition φ leads to the postcondition ψ". The distinction between preconditions and postconditions corresponds to the distinction between input and output in input-output logic [14].

Example: We will describe the evaluation using an example from [9], which is a fragment of the law that regulates collection and testing of blood donations.

(1) Except as specified in (2), every donation of blood or blood component must be tested for evidence of infection due to Hepatitis B.

(2) You are not required to test donations of source plasma for evidence of infection due to Hepatitis B.

Statement (1) conveys an obligation to test donations of blood or blood component for Hepatitis B, and (2) conveys a permission not to test specific types of donations. We represent the two statements above as follows:

- $1 : bb(u) \wedge d(x) \wedge \neg \mathrm{says}_{\{2\}}(\neg \mathcal{O}_u test(x)) \rightsquigarrow \mathcal{O}_u test(x)$, and
- $2 : bb(v) \wedge d(y) \wedge sp(y) \rightsquigarrow \neg \mathcal{O}_v test(y)$

The predicates are understood as follows. $bb(u)$ is true iff u is a bloodbank, $d(x)$ is true iff x is a donation, $sp(y)$ is true iff y consists of source plasma, and $test(x)$ is true iff x is tested for Hepatitis B. In the obligation, the subformula $\mathrm{says}_{\{2\}}(\neg \mathcal{O}_u test(x))$ is understood as "u is not obligated to test x according to statement (2)".

Objects	Predicates	Utterances
o, o_1,	$bb(o)$, $d(o_1)$, $sp(o_1)$, $test(o_1)$	$\mathrm{says}_{\{2\}}(\neg \mathcal{O}_o test(o_1))$
o_2	$bb(o)$, $d(o_2)$, $\neg sp(o_2)$, $\neg test(o_2)$	$\mathrm{says}_{\{1\}}(\mathcal{O}_o test(o_2))$

Table 1: A state and its utterances

Regulatory statements are evaluated with respect to states, which supply valuations of predicates used in the statements, and assignments of objects to variables. If the precondition of a statement is true, the postcondition, with variables substituted by their respective object assignments, is *uttered*. Table 1 shows a state of a bloodbank augmented with utterances. There are three objects – o is a bloodbank, o_1 is a donation of source plasma, and o_2 is a non-source plasma donation. We define conformance as the satisfaction of all obligations that are derived as utterances. Thus the state does not conform to the regulation, since o_2 is not tested.

Evaluating the regulation: We say that a statement id depends on a statement id' if $\mathrm{says}_{Id}\, \psi$ with $id' \in Id$ is used in the precondition of id. If dependencies are acyclic, evaluation is performed in the dependency order; in the case of cycles, a least fixed point is computed. To evaluate the example, we first consider permission $2 : bb(v) \wedge d(y) \wedge sp(y) \rightsquigarrow \neg \mathcal{O}_v test(y)$. Since the precondition of statement (2) is true for the assignment of v to o and y to o_1, we have the utterance $\mathrm{says}_{\{2\}}(\neg \mathcal{O}_o test(o_1))$. However, since o_2 is not a donation of source plasma, there is no corresponding utterance. Now consider the formula $\mathrm{says}_{\{2\}}(\neg \mathcal{O}_u test(x))$ in the antecedent of (1). To evaluate it, we look for a set of utterances U that make the formula provable, that is, $\vdash \bigwedge U \Rightarrow \mathrm{says}_{\{2\}}(\neg \mathcal{O}_u test(x))$. In Table 1, there is an annotation that makes this implication a propositional tautology when u is assigned to o and x to o_1. This lets us, in turn, to produce the utterance $\mathrm{says}_{\{1\}}(\mathcal{O}_o test(o_2))$.

3 Discussion

In this section, we discuss how various constructs from the literature are expressed in our framework. In Section 3.1, we discuss access control examples. Section 3.2 discusses conformance in the presence of iterated deontic modalities [16]. Our approach provides a partial analysis, and we argue that it suffices for access control applications.

3.1 Access Control

We discuss two access control examples in this section. The first example highlights an important restriction of the policies in Section 2.2, i.e., a policy lets us conclude what has been said, but not what actually happens. The second example illustrates how the delegation operator of [13] can be defined in our framework.

Example 1 [10]: Consider a file-access scenario with an administrating principal (A), a user (B), a file ($\mathrm{file1}$), and the following policy:

1. If A says that $\mathrm{file1}$ should be deleted, then this must be the case.
2. A trusts B to decided whether $\mathrm{file1}$ should be deleted.
3. B wants to delete $\mathrm{file1}$.

We introduce a new principal F for the file system. The set U of utterances (U) obtained at the fixed point is $\{\mathrm{says}_{l(F)}\,\mathcal{P}_A\,\mathrm{says}_{l(A)}\,\mathcal{O}_F(\mathrm{delfile1}),\ \mathrm{says}_{l(A)}\,\mathcal{P}_B\,\mathrm{says}_{l(B)}\,\mathcal{O}_F(\mathrm{delfile1}),\ \mathrm{says}_{l(B)}\,\mathcal{O}_F(\mathrm{delfile1})\}$. In the first utterance, the file system F says that A is permitted to require it (F) to delete $\mathrm{file1}$. The second utterance is the delegation from A to B, and the third utterance is B's wish to delete $\mathrm{file1}$. Using **A5**, we will conclude that $U \vdash \mathrm{says}_{l(F)}\,\mathcal{O}_F(\mathrm{delfile1})$. In other words, the system requires itself to delete $\mathrm{file1}$.

Our analysis differs in an important way from [10]. We do not conclude that $\mathrm{file1}$ is actually deleted, i.e., $U \nvdash \mathrm{deletefile1}$. In fact, we can show that there is no policy (as defined in Section 2.2) that lets us make this conclusion. In some cases, it may be warranted to assume/axiomatize self-conformance, i.e., $(\mathrm{says}_{l(F)}\,\mathcal{O}_F(\varphi)) \Rightarrow \varphi$. However, conflicting self-imposed requirements would make U inconsistent.

Example 2: The delegation operator of [13] has a compelling definition in our framework. The syntax (in [13]) for delegation is "x delegates $(\varphi)^d$ to y", where d is the depth of delegation. We define the schema $\mathrm{ps}(\varphi, x, d)$, where x is used to generate variable names, and $d \in N$:

- $\mathrm{ps}(\varphi, x, 1) = \mathcal{P}_{x_1}\,\mathrm{says}_{l(x_1)}\,\varphi$
- $\mathrm{ps}(\varphi, x, d) = \mathcal{P}_{x_d}\,\mathrm{says}_{l(x_d)}(\varphi \wedge \mathrm{ps}(\varphi, x, d-1))$, for $d > 1$

The statement "A delegates $(\mathrm{delfile1})^2$ to B" is interpreted as follows: A says $\mathrm{delfile1}$ if B says it or anyone that B trusts says it. Suppose, in addition, that B delegates $(\mathrm{delfile1})^1$ to C, and C says $\mathrm{delfile1}$. We express this with the following rules:

1. $(x_2 = B) \rightsquigarrow \mathrm{ps}(\mathrm{delfile1}, x, 2)$
2. $(y_1 = C) \rightsquigarrow \mathrm{ps}(\mathrm{delfile1}, y, 1)$
3. $\top \rightsquigarrow \mathrm{delfile1}$

If $1 \in l(A)$, $2 \in l(B)$ and $3 \in l(C)$, we will derive A says $\mathrm{delfile1}$. Further redelegations by C (by modifying statement 3) will not be attributed to A.

In [13], a representation statement is used to allow transfers *without consuming delegation depth*. If C represents B on $\mathrm{delfile1}$, then C is permitted the same redelegation as B. In our approach, delegation is just a special kind of representation. A delegates $(\varphi)^d$ to B iff B represents A on "delegating $(\varphi)^{d-1}$ to anyone". If C represents B on "delegating $(\varphi)^{d-1}$ to anyone", then she represents A as well.

Our approach can capture more complicated cases of delegation. For example, A may not wish to trust C to the same extent as B. Informally, we can express this lack of trust by permitting B to delegate, if she does not delegate to a higher depth. Such conditional delegations cannot be expressed in the formalism in [13]. Note however that the restrictions in [13] are motivated from the perspective of efficiency, while the focus here is on expressive power. Exploring restrictions on rules (for efficiency) is an interesting problem for further research.

3.2 Iterated Deontic Modalities

The main purpose for our extension of the logic in [9] is to provide a (partial) analysis of iterated deontic modalities [16], i.e., sentences of the form "required to allow x".

Example 1 (based on [16]): consider

(3) The owners of parking lots ought to forbid parking near the entrance.

We analyze this sentence as follows: "The owners of parking lots ought to (introduce laws that) forbid parking near the entrance.". In other words, (3) is an obligation to introduce a prohibition. If the owner introduces such a law, then the person parking is viewed as non-conformant, but it is the owner that needs to conform to (3). We can represent (3) in logic as follows:

$3 : owner(x) \wedge person(y) \rightsquigarrow \mathcal{O}_x \,\mathrm{says}_{l(x)}\, \mathcal{O}_y \neg pne(y, x)$

Here $owner(x)$ denotes the owner of a parking lot, $person(y)$ is a person parking a car, and $pne(y, x)$ denotes y parking near the entrance of the lot owned by x.

Let us assume a state where $\{owner(A), person(B), pne(B, A)\}$ hold. Suppose first that A does not introduce any laws, i.e., $l(A) = \emptyset$. The computed utterances are $\{\mathrm{says}_{\{3\}}\, \mathcal{O}_A \,\mathrm{says}_{\emptyset}\, \mathcal{O}_B \neg pne(B, A)\}$. Here, A does not conform to $\{3\}$, because there is an obligation that $l(A)$ does not satisfy; however, B conforms. Now suppose that A introduces the law $2 : person(y) \rightsquigarrow \mathcal{O}_y \neg pne(y, A)$. $l(A) = \{2\}$. The computed utterances now are: $\{\mathrm{says}_{\{3\}}\, \mathcal{O}_A \,\mathrm{says}_{\{2\}}\, \mathcal{O}_B \neg pne(B, A), \mathrm{says}_{\{2\}}\, \mathcal{O}_B \neg pne(B, A)\}$. Here A conforms to $\{3\}$. However, now B does not conform. We have thus captured the situation where the statement (3) conveys an obligation to A and if A conforms, the embedded obligation is conveyed to B.

Example 2: As we mentioned, our approach provides only a partial analysis of iterated modalities. Consider the following example:

(4) You are required to allow a patient to see his records.

By our analysis, (4) is an obligation on the hospital to provide a permission. Let us suppose that a hospital introduces such a permission in its policy. Has it conformed to (4)? The problem arises in distinguishing between *claimed permission*, and *actual permission*. A hospital claims that it permits a patient to see his records, by making an appropriate rule. On the other hand, a hospital actually permits a patient to see his records, by taking an action, e.g., sending the records via mail. Due to the practical difficulties, we focus on claimed permission, and leave open the problem of analyzing actual permission. In access control systems, permitted actions (other than statements) are in the control of the system. Every permitted request can be facilitated by the system, and we assume that the system facilitates accordingly. To our knowledge, this assumption of facilitation is common to all access control systems.

4 Conclusions

We have motivated and described an access control logic that uses the interaction between saying and deontic modalities. We proposed two axioms to characterize the interaction (Section 2.1), and showed how these axioms could be incorporated into a logic programming approach (Section 2.2). In Sections 1 and 3, we discussed how various constructs that have been studied separately are unified in our formalism.

Logic programming has been popular in access control [13, 7, 4]. The formalism that we adopted (Section 2.2) provides a way to integrate the logic programming approaches with the logics of saying, i.e., by evaluating saying using provability. However, the provability tests can be expensive, and it is of interest to identify tractable fragments. The logic programming restriction to Horn clauses, and the techniques in [8, 12] suggest some directions toward this end.

References

[1] M. Abadi. Logic in access control. In *Proceedings of the Symposium on Logic in Computer Science*, 2003.

[2] M. Abadi. Access control in a core calculus of dependency. *Electronic notes in Theoretical Computer Science*, 172:5–31, 2007.

[3] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

[4] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information Systems Security*, 6(1):71–127, 2003.

[5] G. Boella and L. van der Torre. Permissions and obligations in hierarchical normative systems. In *Proceedings of the 9th international conference on AI and law*, 2003.

[6] A. Cirillo, R. Jagadeesan, C. Pitcher, and J. Riely. Do as I SaY! Programmatic access control with explicit identities. In *20th IEEE Computer Security Foundations Symposium*, 2007.

[7] J. Crampton, G. Loizou, and G. O. Shea. A logic of access control. *The Computer Journal*, 44(1):137–149, 2001.

[8] N. Dinesh, A. Joshi, I. Lee, and O. Sokolsky. Checking traces for regulatory conformance. In *Proceedings of the Workshop on Runtime Verification (to appear)*, volume 5289 of *LNCS*, pages 86–103, 2008.

[9] N. Dinesh, A. Joshi, I. Lee, and O. Sokolsky. Reasoning about conditions and exceptions to laws in regulatory conformance checking. In *Proceedings of the Conference on Deontic Logic in Computer Science*, 2008.

[10] D. Garg and M. Abadi. A modal deconstruction of access control logics. In *Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, 2008.

[11] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *19th IEEE Computer Security Foundations Workshop*, 2006.

[12] G. Governatori and A. Rotolo. Bio logical agents: Norms, beliefs, intentions in defeasible logic. *Autonomous Agents and Multi-Agent Systems*, 17(1):36–69, 2008.

[13] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation logic: a logic-based approach to distributed authorization. *ACM Transactions on Information and System Security*, 6(1):128–171, 2003.

[14] D. Makinson and L. van der Torre. Input/output logics. *Journal of Philosophical Logic*, 29:383–408, 2000.

[15] D. Makinson and L. van der Torre. Permissions from an input/output perspective. *Journal of Philosophical Logic*, 32(4), 2003.

[16] R. B. Marcus. Iterated deontic modalities. *Mind*, 75(300), 1966.

[17] L. T. McCarty. A language for legal discourse - i. basic features. In *Proceedings of ICAIL*, 1989.

[18] M. Sergot, F.Sadri, R. Kowalski, F.Kriwaczek, P.Hammond, and H. Cory. The british nationality act as a logic program. *Communications of the ACM*, 29(5):370–86, 1986.

[19] G. H. von Wright. Deontic logic. *Mind*, 60:1–15, 1951.

Integrating Contract-based Security Monitors in the Software Development Life Cycle*

Alexander M. Hoole[1], Isabelle Simplot-Ryl[2], Issa Traore[1]

[1]Dept. of Electrical and Computer Engineering
University of Victoria
P.O. Box 3055 STN CSC
Victoria, B.C. V8W 3P6
CANADA

[2]LIFL CNRS UMR 8022/INRIA Lille-Nord Europe
Universite de Lille I, Cite Scientifique
F-59655 Villeneuve d'Ascq Cedex
FRANCE

E-mail: alex.hoole@ece.uvic.ca isabelle.ryl@lifl.fr itraore@ece.uvic.ca

Abstract

Software systems, containing security vulnerabilities, continue to be created and released to consumers. We need to adopt improved software engineering practices to reduce the security vulnerabilities in modern systems. These practices should begin with stated security policies and end with systems which are quantitatively, not just qualitatively, more secure. Currently, contracts have been proposed for reliability and formal verification; yet, their use in security is limited. In this work, we propose a contract-based security assertion monitoring framework (CB_SAMF) that is intended to reduce the number of security vulnerabilities that are exploitable, spanning multiple software layers, to be used in an enhanced systems development life cycle (SDLC).

1. Introduction

Security has always been a hybrid of art and science as throughout history humans have attempted to protect valuable assets. Our modern information driven society has placed an increased value on data and the transfer and storage of information. More recently, in the last decade, industry and academia have pushed for more secure solutions for information technology assets and facilities as we have equally seen a rise in malicious hacking and security threats.

Many different approaches have been presented recently toward solving the problem of weak security; however, we obviously have not yet found a solution since security related attacks continue to persist.

Gary McGraw identifies three trends that have a large influence on the growth and evolution of the software security problem [13]. First, *connectivity* to the Internet has increased the number of attack vectors and the ease of which an attack can be made. Second, *extensibility* of software is allowing systems to grow in an incremental fashion which potentially adds new security vulnerabilities to existing systems. Lastly, the extensive increase of software *complexity* in modern information systems leads us to a greater number of vulnerabilities. These three trends will continue and lead us to one, hopefully obvious, conclusion. Security and dependability vulnerabilities must be resolved during design and testing before being released to the general public.

Recently, we have observed a promising shift in industry and academia to reduce security vulnerabilities during the software development life cycle (SDLC), rather than attempt to patch the problem after software is shipped [4, 8, 9, 13]. If we can reduce *security defects* early in the SDLC we reduce not only the number of vulnerabilities but also the risk of attack.

While there are areas being researched which target specific areas of security during the systems development life cycle (SDLC), a methodology for testing security across multiple software layers is still lacking. We propose a contract-based security assertion monitoring framework (CB_SAMF) that is intended to reduce the number of security vulnerabilities that are exploitable, spanning multiple software layers, to be used in an enhanced SDLC.

*This work is partially supported by CPER Nord-Pas-de-Calais/FEDER Campus Intelligence Ambiante.

The following section will review SDLC and how it relates to security, Section 3 discusses modeling techniques related to security, Section 4 introduces our proposed approach, Section 5 discusses our contract model, Section 6 expands on the benefits of contracts for security, and Section 7 provides our concluding remarks.

2 SDLC and Security

Security policy documents are often used by organizations to specify the laws, rules, practices, and principles that govern how to manage, protect, and transfer sensitive information. These policy documents represent a corner stone from which software requirements can be built. Requirements in turn drive most modern software/system development life cycles. During the SDLC there are many opportunities to reduce security vulnerabilities.

A SDLC is typically an iterative and recursive process which clearly identifies the stages that should lead a successful software project through its entire development life cycle. We are interested with integrating security into every phase of the SDLC. In fact, several tools and methodologies have already begun to integrate themselves accordingly. We believe, however, that there is a great deal of work remaining in this area.

The SDLC is still lacking models, methods, and tools that assist in creating more secure and reliable software products. The audience for this work includes individuals and teams fulfilling the following roles during a SDLC: analyst, architect, developer, tester, maintainer, user, and support. Essentially, all of the development-related stake holders in the SDLC.

Recently Serpanos and Henkel asserted that a unified approach to dependability and security assessment will let architects and designers deal with issues in embedded computing platforms [18]. The observation that security and dependability are interrelated is an important one. Serpanos and Henkel differentiate the two based on security flaws being problems that are exploited on purpose, while flaws which are exploited by accident would be qualified as dependability problems. It would be interesting to have a framework that can support both dependability and security. Thus, we have kept dependability in mind while designing our framework; however, we focus on security vulnerability monitoring since it is our primary concern.

The goal of our research is to create new methods, models, and tools that integrate into the phases of the SDLC to create more secure software. We cannot always depend on the consumer to have sufficient protection mechanisms in place on their systems.[1] We need to take a more active role during development to ensure software ships fewer security vulnerabilities.

A modified form of the SDLC is depicted in Figure 1 showing how various security activities can be integrated into the iterative and recursive SDLC. Existing SDLC hybrids integrate some of the steps identified in Figure 1 such as those put forward by CERT, Microsoft's Michael Howard and Steve Lipner [9], and others. Nothing has been identified to date that guarantees security in software systems; however, our aim is to help reduce the risk associated with security vulnerabilities.

3 Modeling

For many years software developers have been using methodologies meant to simplify and standardize the SDLC. One notation that has met with a great deal of success, in several methodologies, is the Unified Modeling Language(UML). UML does not handle all analysis, design, and implementation requirements for all projects. For example, UML is a natural fit for most object oriented languages; however, not all projects demand an object oriented approach. Projects that require high performance, and a low memory footprint, are typically implemented in non-object oriented languages such as C.

Several diagrams that are used in UML are useful in the broader spectrum of all software design projects. For instance, use case diagrams are very useful in identifying the main functions of a software artifact. In fact, use cases provide the earliest opportunity to identify security risk in a new SDLC for a given application (other than general risk analysis).

Recently, a more modern addition to use cases, called misuse cases, has been created. Misuse cases, also known as abuse cases, can be used during requirements analysis [1, 7, 17] leading to a more complete understanding of potential security risks that need to be mitigated. Hope, McGraw, and Antón also mention that misuse cases can be over-used and can lead to identification of a fairly large set of misuse cases that may have little impact on security [7]. With the knowledge of subject matter experts and security analysts these misuse cases need to be prioritized to balance risk and cost. During development many risks can be completely mitigated based on the early warnings of the misuse cases. We must recognize, however, that the probability of a particular misuse may not be completely understood and that some risks may not be identified until later in the SDLC. It would be useful to have a mechanism that can identify these security threats in the code and allow for a monitoring system to be implemented to capture and trace any possible misuse.[2]

[1] Consumers often employ intrusion detection systems, firewalls, and other products to help reduce security risks.

[2] An example of such an approach would be to use the output of static analysis security tools as the basis of misuse case creation.

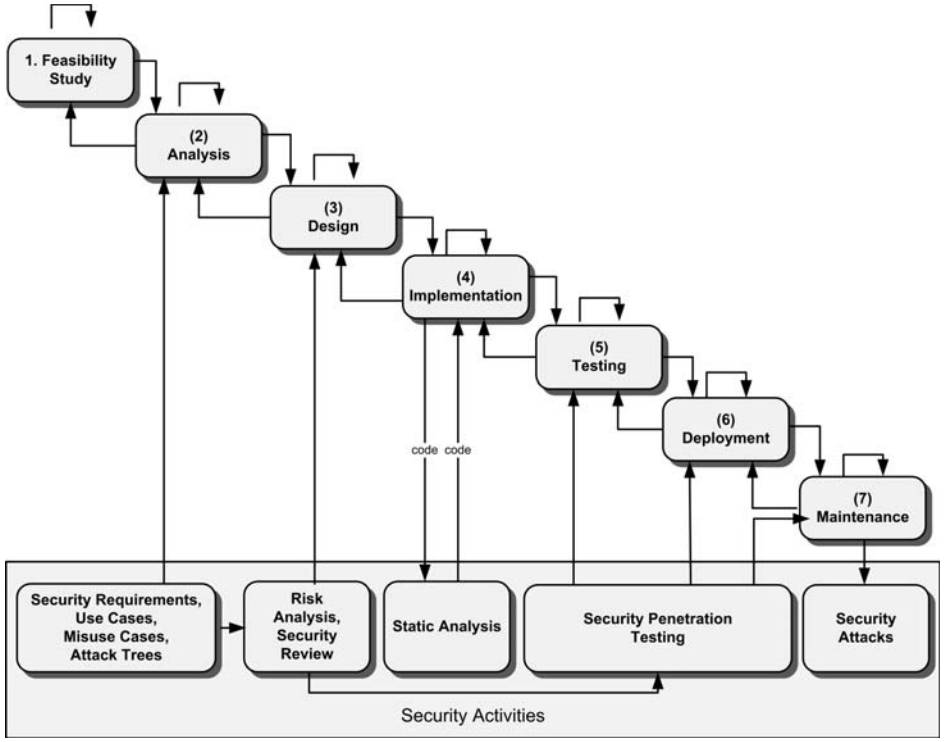


Figure 1. Security activities integrated into the typical waterfall SDLC. Regular SDLC steps are numbered and linked in diagonal. Security activities are shown horizontally.

Misuse case diagrams can be used to expose a wide variety of threats including privacy violation, denial of service, privilege escalation, identity or information theft, and network based attacks. As with use case diagrams, misuse case diagrams are continually reviewed and revised throughout the SDLC. The components that make up a misuse case are documented already in [1, 7, 17].

Once misuse cases have been identified we can then proceed with the identification of security violation scenarios. One technique for identification of these violation scenarios is the use of an attack tree. Each depth first traversal of an attack tree will identify possible violation scenarios [15].

4 Proposed Approach

Now that we have discussed some of the methods for identifying potential vulnerabilities, we propose a model for monitoring applications for security violations during the middle phases of the SDLC which also allows for the collection of forensic data based on the prioritized security risks identified earlier in the SDLC.

This monitoring framework can be integrated early during SDLC. In Figure 2, we depict how the security policy document is used as part of the processes identifying the security requirements. Security requirements are then used during the identification of misuse cases (along with normal use cases) that are intended to identify potential vulnerabilities. Once prioritized, these misuse cases can then drive the creation of attack trees which further identify intrusion scenarios. The intrusion scenarios can then be used during design and testing to create sequence diagrams and associated test cases. Finally, during implementation, sequence diagrams can be generated which identify security vulnerabilities (for example, system/function calls that have known vulnerabilities). Once a vulnerability has been identified, a ”contract” can be created using assertions and additional rules to guard against, or verify, a given vulnerability. These contracts can then in turn be used to generate security probes that are used during execution to track forensic data in our monitoring framework (CB_SAMF).

Consideration should be given as to whether or not output formats from existing tools, such as static analysis tools, may be translated into a format that may be used by the assertion monitoring framework.

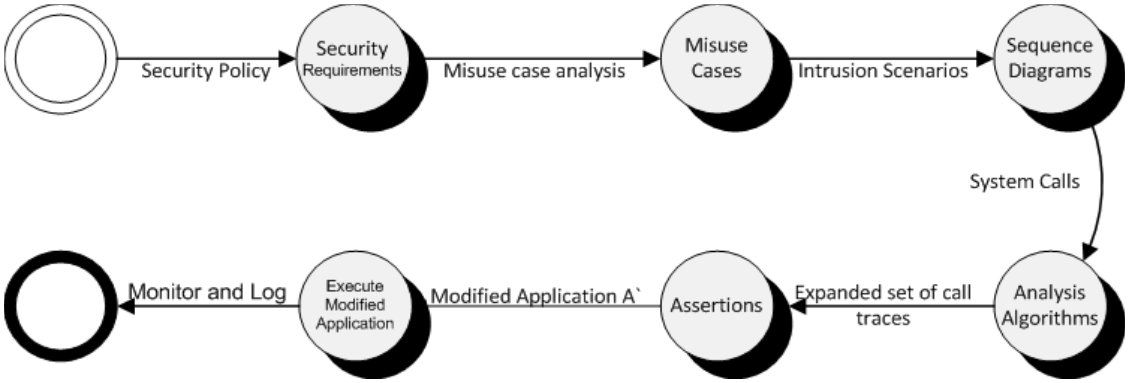


Figure 2. System flow diagram leading to the use of contracts and monitoring probes.

Ultimately the focus of the initial work will be on the last three nodes of Figure 2 by creating and consuming a contract, generating the assertion probes, monitoring assertions, and reacting appropriately using the monitoring framework.

5 Contract Model

The notion of a contract used in software engineering is not a new idea [6, 10, 11, 14]. When used for security, however, we must look outside of the basic preconditions and postconditions that are often used when implementing systems using contracts and look carefully at what properties need to be specified in a contract to improve security. Historically, the precondition specifies when it is appropriate to call a particular feature (function/method), while a postcondition specifies what is true after a particular feature is called (what has been accomplished by the function/method). [3]

Our definition of contract needs to bind the caller and callee to deal with additional properties involving timing, property values, and other events.[4] For example, a contract that is specified for a supplier *X* is consumed by a consumer *Y* guarantees that *X* has fulfilled the postcondition(s), provided that *Y* has satisfied the precondition(s). Thus, the contract provides protection for both parties. The consumer is protected from the supplier since the postconditions have been guaranteed by the supplier. The supplier is protected from the consumer since the preconditions have been guaranteed by the consumer.

Contracts, as proposed by Meyer, are not suitable for security monitoring.[5] The require, guarantee, and references fields of the contract, that correspond to the pre, post, and invariants, do not handle all of the necessary attributes of security defects. In particular, we would propose the addition of several new contractual fields including *context*, *history*, and *response*. Context is required since the basic reliability contract above does not factor environmental influence. History is required since security vulnerabilities are often complex and are sometimes the result of a series of actions which may occur in parallel. Both context and history can be useful when dealing with DoS and race-condition vulnerabilities. Finally, response is required so that we can choose how a particular assertion is handled when an exploitation is detected. We desire the ability to deal with security assertion failures, not just detect them as would be the case if we used the form of contract proposed by Meyer.[6]

Our form of contract includes the following fields:

- Requirements - in the form of preconditions (PRE)
- Guarantees - in the form of postconditions (POST)
- References - in the form of invariants (INV)
- Context - in the form of relevant environmental information (CONT)
- History - in the form of some knowledge keeping construct (HIST)

[3] Many pre and postconditions are more to do with robustness than security.

[4] The definition of **binding contract**: The legal agreement between two or more entities to perform and/or not perform a set of actions.

[5] For example,under normal contracts, a false precondition does not guarantee that the system will not process the input. It may still allow certain types of attacks such as buffer overflows to continue.

[6] The concept of resumption and organized panic for exception handling, used by Meyer, could also fall under our broader response category [14].

- Response - in the form of a reactive measure (RESP)

Work done by Barringer *et al* on program monitoring and rule-based runtime verification has exposed interesting results [2, 3]. Specifically, the work on linear temporal logic (LTL) and program states has been core to several attempts towards runtime verification and is a promising candidate for the notation of our contracts.

Each contract (*C*) will contain a breakpoint (*B*) and one or more assertions (*A*). A breakpoint identifies a monitoring location or symbol in the target application. For example, a contract should be able to specify a target function in a program which affects the state of an assertion. The assertion is a rule which must remain true at the breakpoint. Each assertion has associated with it zero or more of the security contract extensions (*E*) mentioned above (context, history, and response). An assertion can take on one of the following three forms: precondition (*PRE*), postcondition (*POST*), or invariant (*INV*). We do not represent the assertions types separately since they all take the same form. Each assertion is composed of zero or more rules (*R*), relating to the target (remember the breakpoint *B*), and zero or more monitors (*M*). The rules, monitors, and extensions are individually named (*N*). A rule specifies a property of the state of the program which needs to remain true, while a monitor enforces one or more rules. The quantifiers *min* and *max* represent liveness and safety properties respectively and are important for the boundary cases of a monitor trace. The body of every rule and monitor is specified as a boolean valued formula of the syntactic category *Form*.[7] Therefore, each contract may be instantiated using the following grammar[8]:

$C := B\ (A\{E\})\ \{A\{E\}\};$
$E := \{CONT\} \mid \{HIST\} \mid \{RESP\};$
$A := \{R\}\{M\};$
$R := \{\underline{\text{max}}|\underline{\text{min}}\}\ N(T_1 x_1, ..., T_n x_n) = F;$
$M := \underline{\text{mon}}\ N = F;$
$T := \underline{\text{Form}} \mid \textit{primitive type};$
$B := \textit{symbol} \mid \textit{HEX address};$
$F := exp|\underline{\text{true}}|\underline{\text{false}}|\neg F|F_1 \wedge F_2|F_1 \vee F_2|F_1 \rightarrow F_2|\odot F|\ominus F|$
$\quad F_1 \cdot F_2|N(F_1, \ldots, F_n)|x_i;$
$CONT := \underline{env}\ N \mid \underline{res}\ N;$
$HIST := \underline{trace}\ N \mid \underline{runningsum}\ N \mid \underline{runningavg}\ N;$
$RESP := \underline{core}\ N \mid \underline{term}\ N \mid \underline{kill}\ N \mid \underline{log}\ N;$

When defining rules, the max prefix indicates that a given rule defines a safety property and min indicates that a rule is a liveness property [3, 16].[9] We have also tentatively defined possible extended behaviors for context, history and response elements and may extend these in the future. Context may specify environmental or resource information (external to the program) which is needed by the contract. History may contain trace data or statistically relevant information for the contract. Finally, response may specify an action to perform an assertion is violated.[10]

From this definition it is possible to use multiple separate monitors or redirect multiple rules to the same monitor.

6 Benefits of Contract for Security Monitoring

Targeting the identification, verification and removal of security vulnerabilities from systems is not a trivial task. We chose the notion of contracts for an assertion framework so that we can state precise properties about a system without having to

[7] This notation is derived from linear temporal logic (LTL) and is inspired by the EAGLE framework that was proposed by Barringer *et al* [2, 3].

[8] Each line is a Extended Backus-Naur Form (EBNF) production. Following is a simplified description of EBNF notation that we have used:
:= meaning "is defined as"
| meaning "or"
, meaning concatenation (used to separate items in a sequence)
{ } meaning zero or more times
{ }- meaning one or more times
[ ] meaning optional item
( ) meaning grouping
; marks the end of a rule

[9] Safety properties state that if a behavior is unacceptable any extension of that behavior is also unacceptable. Liveness properties state that for a given requirement, and any finite duration, the behavior can always be extended such that it satisfies the requirement[16, 12].

[10] Possible responses include the following: core=produce a core dump, term=terminate the task, kill=kill the task, log=produce an audit report for the event.

modify the code directly. In order to understand the benefits of these contracts for security monitoring we will briefly discuss a variety of common security vulnerabilities. An example set of common security problems found in systems is as follows:

- Exploitable Logic Error
- Inadequate Concurrency Control
- Weak Dependencies/Altered Files
- Inadequate Parameter Validation Incomplete/Inconsistent
- Inadequate Authentication/Authorization/Identification
- Implicit Sharing of Data and Data Leakage

As we progress with this work we expect that a wide variety of vulnerabilities should be covered by contracts. Exploitable logic errors are difficult to track down; however, if we can identify environmental, historical, or timing information related to the expected behavior, contracts can be written to detect misuse. Parameter validation issues can be handled by our pre and post conditions. Concurrency, accountability, and protocol issues can be tracked through the use of historical, environmental, pre and post conditions. Finally, the addition of historical and environmental assertions should allow us to track vulnerabilities related to weak dependencies and data leakage. Furthermore, to give an idea of the types of attacks we should attempt to counter, a listing of network related attack classes is as follows (derived from [5]):

- Password Stealing
- Bugs and Back Doors
- Protocol Failures
- Exponential Attacks Viruses and Worms
- Botnets
- Social Engineering
- Authentication Failures
- Information Leakage
- Denial-of-Service Attacks
- Active Attacks

Contracts are not suitable for dealing with all types of attacks. For example, password stealing can occur through the use of a network sniffer or through the use of social engineering techniques. The ability of an attacker to passively monitory network traffic will not be prevented through the use of contracts; however, we can use contracts to ensure that security properties of our systems (derived from our initial security policies) are observed. In the case of password stealing, the password should never enter a public network in clear text and the protocol used for authentication should not be subject to replay attacks. These are properties for which we can design contracts.

7 Conclusion

Our enhanced version of contracts provides a novel way to propagate requirements-based security assertions through the SDLC. Some techniques, such as misuse cases, attack trees, and static analysis, are already providing ways of identifying potential vulnerabilities during the early phases of the SDLC; however, these approaches can lead to a high rate of false-positives which consume resources. Our (CB_SAMF) is able to help reduce vulnerabilities in multi-layered systems by not only providing a way to detect if a particular contract is violated, but also provides reactive measures.

References

[1] I. Alexander. Misuse cases: Use cases with hostile intent. *IEEE Software*, 20(1):58–66, 2003.
[2] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Program monitoring with ltl in eagle. *ipdps*, 17:264b, 2004.
[3] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification, 2004.
[4] M. Bishop. *Computer Security: Art and Scienc*. Addison-Wesley, Boston, MA, USA, 2003.
[5] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.
[6] A. Février, E. Najm, and J.-B. Stefani. Contracts for odp. In *ARTS '97: Proceedings of the 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software*, pages 216–232, London, UK, 1997. Springer-Verlag.
[7] P. Hope, G. McGraw, and A. I. Anton. Misuse and abuse cases: Getting past the positive. *IEEE Security and Privacy*, 02(3):90–92, 2004.
[8] M. Howard. Building more secure software with improved development processes. *IEEE Security and Privacy*, 2(6):63–65, 2004.
[9] M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.
[10] J. C. M. Jr. Programming by contract. *Computer*, 29(3):109–111, 1996.
[11] L. Lamport. A simple approach to specifying concurrent systems. *Commun. ACM*, 32(1):32–45, 1989.
[12] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.
[13] G. McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.
[14] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
[15] A. Moore, R. Ellison, and R. Linger. Attack modeling for information security and survivability, 2001.
[16] D. K. Peters and D. L. Parnas. Requirements-based monitors for real-time systems. *SIGSOFT Softw. Eng. Notes*, 25(5):77–85, 2000.
[17] G. Peterson and J. Steven. Defining misuse within the development process. *IEEE Security and Privacy*, 04(6):81–84, 2006.
[18] D. Serpanos and J. Henkel. Dependability and security will change embedded computing. *Computer*, 41(1):103–105, 2008.

Towards verifying contract regulated service composition

Alessio Lomuscio Hongyang Qu Monika Solanki
Department of Computing, Imperial College London, UK

Abstract

We report on a novel approach to (semi-)automatically compile and verify *contract-regulated* service compositions. We specify web services and the contracts governing them as WSBPEL behaviours. We compile WSBPEL behaviours into the specialised system description language ISPL, to be used with the model checker MCMAS to verify behaviours automatically. We use the formalism of temporal-epistemic logic suitably extended to deal with compliance/violations of contracts. We illustrate these concepts using a motivating example whose state space is approximately 10^6 and discuss experimental results.

1 Introduction

Web services (WS) are now considered one of the key technologies for building new generations of digital business systems. Industrial strength distributed applications can be built across organisational boundaries using services as basic building blocks. When services are combined, a significant challenge is to regulate the business interactions between them. In an environment where previously unknown services are dynamically discovered and binded, their composition is usually underpinned by binding agreements or "contracts". Should a contract be broken by one of the parties, "legal remedies" may be applicable in the form of penalties, additional rights to some party, and, possibly, additional penalties with respect to third parties.

Conventionally, contracts have been defined and interpreted using natural languages. In electronic business environments, new formal models and tools are needed to enable the successful enforcement of dynamic contractual agreements between services. While designing a contract-regulated composition, an important aspect is the rigorous analysis of possible execution behaviours of individual services as well as the overall behaviour of the composition. A system made of few localised services may only interact in a small number of ways governed by a limited set of contract clauses. However when several subsystems coordinate in an open environment, the contracts binding them are non-trivial and complex, making it difficult to forsee all the possible executions. Additionally, while trying to comply to their respective contractually defined behaviours, certain components may fail, some may be incapacitated to provide the services in the expected timeline, and others still may have to prioritise certain requests.

In this paper, we propose a novel approach towards the verification of services, where transactions are controlled by binding electronic contracts. Verification of WS is an active topic of research (e.g., see [16, 18]). However it has so far been concerned with checking safety and liveness properties only. Our proposed framework, builds upon existing work in the domain of multi agent systems (MAS) [17, 1]. We take the view that a web service can be modelled as an "agent" [5]. When WS are phrased as a contract-regulated MAS, several properties become worth studying, including various notions of correctness and violations of the contracts during a run, the evolution of the agents' knowledge about themselves, the contracts and the expected peers' behaviours, etc.

The specification and analysis of agent behaviour in a MAS has been widely explored. Several formal models have been investigated to specify formally and unambiguously the behaviour of the system. Many of these are based on modal logic, including temporal, epistemic, and deotic logic. Developments in verification of MAS via model checking techniques [15, 4, 9] has kept pace with the advancement in the specification techniques. Along with temporal languages, it is now also possible to verify a variety of modalities describing the informational and intentional state of the agents.

The above leads us to explore the verification of contract-based WS implemented by means of MAS model checkers. To this end, we propose a verification methodology where services or "contract parties" (CP) are specified using WSBPEL [13]. The contractually correct behaviours for every CP are also specified in WSBPEL. In our approach, a compiler of our design takes as input both these behaviour descriptions, and generates an ISPL program, which is fed to the symbolic model checker MCMAS for verification.

The rest of the paper is organised as follows. In Section 2 we briefly introduce WSBPEL, ISPL and MCMAS. Section 3 introduces a motivating example and some of its key properties. Section 4 presents our proposed framework. Section 5 discusses the implementation of the compiler and Section 6 gives experimental results from verification. We conclude in Section 7.

2 Preliminaries

2.1 MCMAS and ISPL

MCMAS [11] is a specialised model checker for the verification of multi-agent systems. It builds on symbolic model checking via OBDDs as its underlying technique, and supports CTL, epistemic and deontic logic. The current version of MCMAS [10] has the following features: (1) Support for variables of the following types: Boolean, enumeration and bounded integer. Arithmetic operations can be performed on bounded integers. (2) Counterexample/witness generation for quick and efficient display of traces falsifying/satisfying properties. (3) Support for fairness constraints. This is useful in eliminating unrealistic behaviours. (4) Support for interactive execution mode. This allows users to step through the execution of their model.

MCMAS uses ISPL as its input language. A system encoded in ISPL is composed of the environment e and a set of agents $A = \{1, \ldots, n\}$. Each agent $i \in A$ has a set of *local states* L_i and a set of local actions Act_i. The *protocol function* of agent i, $P_i : L_i \rightarrow 2^{Act_i}$, defines for each local state $l_i \in L_i$ the set of actions that are allowed to be executed in l_i. Similarly, the environment has its local states L_e, local actions Act_e and protocol function P_e. The transition relation among local states of agent i is defined by the *evolution function* $Ev_i : L_i \times Act_1 \times \cdots \times Act_n \times Act_e \rightarrow L_i$. The definition of Ev_i suggests that the local actions of an agent can be observed by other agents. The evolution function Ev_e of the environment is defined in the same way.

To reason about the behaviours of agent i with respect to correctness [12], L_i is further partitioned into two disjoint sets: a non-empty set G_i of allowed ("green") states and a set R_i of disallowed ("red") states. In this paper, we use green states to denote the behaviours in compliance with contracts and red states to denote violations, by means of temporal epistemic properties.

ISPL allows user defined atomic propositions over *global states* of the system. A global state is composed of a local state from every agent and the environment. The logic formulae to be checked by MCMAS are defined over the atomic propositions.

2.2 WSBPEL

WSBPEL [13] is a popular and de facto industrial standard for describing service composition. The specification has been elaborately explained in several web service based literature [13]. is highly recommended.

WSBPEL defines a model and an XML based grammar for the orchestration of executable and abstract business processes. A BPEL process defines the interaction between partners. The specification provides the control logic to coordinate arbitrarily complex web services, defined in WSDL. A BPEL process can interact synchronously or asynchronously with its partners, i.e., its clients, and with the services the process orchestrates.

The building blocks for a BPEL process are the descriptions of the parties participating in the process, the data that flows through the process and the activities performed during the execution of the process. Some examples of activities include "receive", "reply", "assign", "sequence" and "wait". WSBPEL also introduces systematic mechanisms for dealing with business exceptions and processing faults. Moreover, WSBPEL introduces a mechanism to define how individual or composite activities within a unit of work are to be compensated in cases where exceptions occur or a partner requests reversal.

3 A Motivating Case Study

In this section we present a composition of services, regulated as a pre-defined contract. The case study was first presented in earlier work on verifying service composition with MCMAS [1]. Here, we focus on the automatic compilation of services from WSBPEL into ISPL.

In the example, the participating contract parties comprise: a principal software provider (PSP), a software provider (SP), a software client (C), an insurance company (I), a testing agency (T), a hardware supplier (H), and a technical expert (E). The high-level workflow of the composition is defined as follows: Client C wants to get a software developed and deployed on hardware supplied by H. To deploy the software, the technical expert E is needed. Components of the software are provided by different software providers. We consider two software providers here: PSP and SP. The components need to be integrated by the providers before the software is delivered to C.

The software integration is carried out by PSP, when SP delivers its component. PSP and SP twice update each other and C about the progress of the software development. Should the client like any changes in the software, he can request them before the second round of updates. Any change suggested by the client after the second update is considered a violation and the client is charged a penalty. The client can recover from this violation by paying the penalty or by withdrawing the request for changes. If PSP and SP do not send their updates as per schedule, this is also considered a violation and they are charged a penalty. Every update is followed by a payment in part by the client C to the PSP. Payment to SP is handled by PSP and is done once the software is deployed successfully.

PSP's obligations:
1. Update *SP* and *C* twice about the progress of the software.
2. Integrate the components and send them to *T* for testing.
3. If components fail, integrate the revised software and send them for testing.
4. Make payment to *SP* after successful deployment of software.

C's obligations:
1. Request changes before the second round of updates.
2. Pay penalty if changes are requested after second round of updates.
3. Make payment to the *PSP* after every update.

Figure 1: Obligations of Contract parties

	Agent	Violation condition	Recovery
1	*PSP*	- does not send messages to *SP* and/or *C* in the first and/or second run of update.	pay penalty charge
2		- does not send payment to *SP*.	no
3	*SP*	- does not send update messages to *PSP* or *C*.	pay penalty charge
4		- does not send its components to *PSP*.	no
5	*C*	- request changes after second update.	pay penalty charge or withdraw changes
6		- does not send the payment to *PSP*.	no
7	*T*	- does not send the testing report to *C*, *PSP* and/or *SP*.	no
8	*H*	- does not deliver the hardware system to *C*.	no
9		- ignores the deployment failure.	no
10	*E*	- does not deploy the software on the hardware system.	no
11	*I*	- does not process the claim of *C*.	no

Figure 2: Agents and their violation conditions.

PSP integrates the components and sends the integrated component to *T* for testing. Results from testing are made available to all the parties, i.e., *PSP*, *SP*, and *C*. If the integration test fails, the components are revised and tested again. Components can be revised twice. If the third test fails, *C* cancels the contract with *PSP*. If the testing succeeds, *C* invokes *I* to get the software insured. *C* then invokes *H* to order the hardware. Finally *C* invokes *E* to get the software deployed. If the software cannot be deployed then the hardware and the components have to be re-evaluated. Components can be revised twice. If the third test fails *C* always cancels the contract with *PSP* and *H*. Figure 1 illustrates the obligations of the *PSP* and *C*.

From the above scenario it can be seen that contracts between services can be usefully employed to illustrate the notion of correctness in behaviour. Any deviation from the behaviour identified in the contract is considered a violation. The contract might in some cases also specify mechanisms for recovering from violations.

The contract between various parties can be violated in many ways. Figure 2 illustrates informally some of the conditions under which some local violations may occur.

4 Verification framework

In this section we discuss our framework for the verification of contracts. Our approach targets two levels of verification:

- conformance of the behaviour of an individual contract party to its contractually correct behaviour.
- conformance of the combined behaviour of all the contract parties to the overall contract.

For the sake of clarity in the figure and the paper, we elaborate on the components of the architecture and the verification methodology, only for contract party C_1. Note that a similar mechanism would be replicated for all the contract parties in the composition.

1. **Natural language contracts:** Conventionally contracts are specified in a natural language. A contract stipulates the obligations of parties entering the contract. It defines behaviours that are considered to be violation of some obligations, and may outline penalties and/or recovery actions from the violations. For verification, a conventional contract is encoded as an *e-contract* in WSBPEL.

2. **Contract party:** A contract party (CP) is a service, that is a first class citizen of the contract regulated composition. The behaviour of a CP is governed by the rights, obligations and violations stipulated in the contract, and agreed to by the CP. The overall fulfillment of a contract depends on the adherence of each CP in the composition to its specified behaviour. In our framework, each contract party is an *agent* with well defined *green* and *red* states corresponding to states of compliance and violation respectively. Our proposed methodology aims to verify the adherence of each agent's behaviour to what has been specified as contractually correct behavior for the agent.

3. **Contract party/agent behaviour:** The behaviour of an agent can be defined in terms of a two-part behaviour: all possible behaviours and contractually correct behaviours. In order to automate the verification, we encode both these behaviours in BPEL. Note that it is possible to describe contractually correct behaviours using a specification language, tailor-made for describing contracts e.g., [14]. However, keeping both these behaviours at the same level of abstraction, provides the system designer with the flexibility needed to combine and compile the behaviours into a model suitable for verification.

   For an agent, we refer to its all possible behaviours as *BPEL-behaviour* and the contractually correct behaviours as its *BPEL-contract*. Note that both the behaviours are inter-dependent and replicate information such as variable and action description for the agent, in their specification.

4. **Compiler:** The compiler is a novel and integral component of our architecture. The compiler takes as input the BPEL-behaviour and the BPEL-contract for an agent and combines them to generate an ISPL program. The compiler parses the BPEL-behaviour to generate a partial model that enumerates the local states but abstracts from defining red and green states. The BPEL-contract is then parsed to enumerate the green/red states for the agent. The internal details of the compiler are illustrated in Section 5.

5. **ISPL and MCMAS:** The ISPL program compiled semi-automatically from the BPEL specification, encodes the overall and desired behaviour of an agent. The program is fed to MCMAS for verification of the agent's behaviour.

5 Implementation

The core component of our framewrok is the compiler that translates a WSBPEL specification into an ISPL program. It generates basic atomic propositions and properties automatically for verification.

Given the two specifications (BPEL-behaviours and BPEL-contracts), we propose a three step methodology to generate the corresponding ISPL program:

1. We represent a BPEL process by an automaton. The BPEL-behaviour is first read into memory, followed by the BPEL-contract. Both behaviours are translated into automata. We use *behaviour automata* to denote the automata representing the BPEL-behaviour and *contract automata* for the BPEL-contract.

2. For each state in the contract automata, we look for its counterpart in the behaviour automata and label it as *green*. We then label all other states in the behaviour automata as *red*. Based on these labels, basic properties specified in temporal-epistemic logic are generated.

3. The labelled behaviour automata and the properties are written to the ISPL file input to the checker.

In what follows, we discuss the methodology in detail.

5.1 Translating BPEL programs into automata

The compiler uses the following rules to do the translation.

- "Assign", "receive", "invoke" and "empty" activities are translated into transitions connecting the respective *source state* and *target state*. A "sequence" activity is translated into a sequence of transitions.

- An "if" activity is translated into two sequences of transitions, one for the *if*-branch and another for the *else*-branch. The first transition in the *if*-branch uses the condition in the "if" activity as its guard, while the first transition in the *else*-branch uses the negation of the condition as the guard. A "while" activity is translated in the same way as an "if" activity except that the target state of the last transition and the source state of the first transition in the *if*-branch are the same.

- "OnMessage" activities and "onAlarm" activities in a "pick" activity are translated into transitions with a common source state.

- A branch in a "flow" activity is translated into a separate automaton. The beginning and the end of these automata are synchronised with the automaton representing the BPEL process. In doing so, we differ from [8], where a "flow" is translated such that: all branches are executed sequentially and all possible permutations are represented as a single automaton.

- Fault handlers and exceptions are translated into transitions as well. The latter transition assigns a specific value to a variable and the guard of the former transition tests if the variable has this value. Other kinds of handlers are dealt with in the same way. Theoretically, in every state where an exception could happen a copy of the exception/handler transition is produced using this state as its source state (note that these copies have the same target state). Thus one transition would be replicated many times. In practice, however, we have a succinct way to implement it due to the flexibility of ISPL, as discussed later.

As remarked in the literature review, much work has been appeared on translating BPEL into model checkers' input languages, e.g., [8, 7, 2]. However, only few of them can process all BPEL structures. A detailed discussion can be found in [2].

5.2 Colouring the model

We use the green and red of labelling in ISPL code to differentiate between contractually correct and incorrect behaviours, as shown in Figure 3. This is possible because the BPEL-contract specification defines behaviours

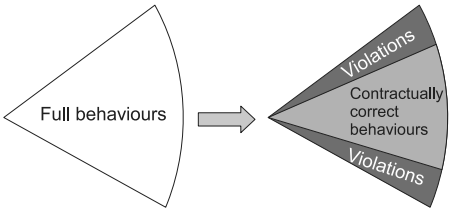


Figure 3: Labelling behaviours

included also in the BPEL-behaviour specification. Labelling the states in the behaviour automata is done as follows:

1. The initial state of a behaviour automaton is labelled as green.
2. For every transition in the contract automata, we find the same transition in the behaviour automata and label its target state as green.
3. For all states that are not green, we label them as red.

We do not look for matched states directly because the states are named in a numerical way and, therefore, the same state in the behaviour automata and the contract automata might have different names. However, transitions get their name from the BPEL activities, each of which has a unique name.

After the labelling process finishes, the compiler encodes three kinds of atomic propositions, which are used to define basic formulae to be checked in the following way. For each BPEL process p, we define

- an atomic proposition p_{green} holding in all green states of the process;
- an atomic proposition p_{end} holding in the last state of process p;
- an atomic proposition p_{red_i} holding in the corresponding red state i.

Two kinds of basic properties are generated based on the atomic properties. For each BPEL process p, define

$$E\ (p_{green}\ U\ p_{end}). \tag{1}$$

This property specifies that p has a way to conduct a whole run in compliance with its contract obligations. For each atomic proposition $p_{red} \in \{p_{red0}, p_{red1}, \ldots\}$, define

$$EF\ p_{red}. \tag{2}$$

This property represents a test to check whether a agent may violate its contractual behaviours.

The above properties verify the basic behaviours of contract parties. More properties can be manually added to the automatically generated ISPL code in order to test other interesting behaviours (see below).

5.3 Generating an ISPL program

Once the behaviour automatas are labelled, they are ready to be written to an ISPL file for verification. Each automaton is mapped to an agent in the file. Let $\mathcal{A} = \{1, \ldots, n\}$ be the set of automata and $A = \{1, \ldots, n\}$ the set of agents. Here we only enumerate the key steps to generate an agent $i \in A$ from an automaton $\mathcal{A}_i \in \mathcal{A}$.

1. Local states generation. A local state $l \in L_i$ is a valuation for the set of *local variables* Var_i. Thus, the generation of L_i is performed through the generation of Var_i. If $\mathcal{A}_i$ is generated from a BPEL process p, then

   $$Var_i = Var_p \cup \{state\},$$

   where Var_p is the set of variables defined in p and *state* is an additional enumeration variable. Each value of *state* represents a unique state of $\mathcal{A}_i$. If $\mathcal{A}_i$ is a "flow" branch in p, then

   $$Var_i = Var'_p \cup \{state\},$$

   where $Var'_p \subseteq Var_p$ is the set of variables used by $\mathcal{A}_i$. In order to reduce the agent's state space, the compiler monitors the usage of every variable $v \in Var_p$. If v is never read by any transitions in $\mathcal{A}_i$, then it is discarded.

2. Local actions generation. Act_i is obtained from the transitions of $\mathcal{A}_i$. Each transition is mapped into an action; additionally if two transitions have the same name, they are mapped into the same action.

3. Protocol generation. Let $l(state)$ be the value of variable *state* in state $l \in L_i$ and E_l the set of allowed actions in l. For any transition t whose source state is represented by $l(state)$, the action to which t is mapped is included in E_l. Obviously, two states $l_1, l_2 \in L_i$ have the same set of allowed actions if $l_1(state) = l_2(state)$.

4. Evolution function generation. Each transition in $\mathcal{A}_i$ is translated to an evolution item. For a transition t with source state s_1, target state s_2, and guard c, the evolution item is defined to be of the following form:

   $$\text{state}{=}s_2 \text{ if state}{=}s_1 \text{ and c and Action}{=}\text{t}.$$

   This item means that if in the current state, the variable *state* has value s_1 and the guard c is satisfied, the execution of t makes agent i move to a state where *state* has value s_2. If t is synchronised with another transition t' in the automaton $\mathcal{A}_j \in \mathcal{A}$, then the evolution item is

   $$\text{state}{=}s_2 \text{ if state}{=}s_1 \text{ and c and Action}{=}\text{t and } \mathcal{A}_j\text{.Action}{=}\text{t'}.$$

   If t assigns a value *expr* to a variable v, the assignment is translated on the left side of "if", i.e.

   $$\text{state}{=}s_2 \text{ and v}{=}expr \text{ if } \cdots.$$

   If there are multiple copies of t, e.g., t represents a fault handler, we use the following form to specify an evolution item for all copies:

   $$\text{state}{=}s \text{ if (state}{=}s_1 \text{ or state}{=}s_2 \text{ or } \ldots) \text{ and c and Action}{=}\text{t and } \cdots,$$

   where s_1 and s_2 are the source states of these copies and s is their target state. If t is allowed in all states, the above form can be simplified to

   $$\text{state}{=}s \text{ if c and Action}{=}\text{t and } \cdots.$$

6 Experimental Analysis

We evaluated the compilation and verification mechanism on the case study illustrated in Section 3. We represented the composition in terms of a WSBPEL orchestration. The following BPEL code represents the full behaviour of the client C, when receiving updates from PSP and SP. Note that for brevity, only essential information is shown. The BPEL-contract is the same as BPEL-behaviour except that it defines only contractually correct and therefore limited behaviours.

```
<pick name="Update1">
  <onMessage partnerLink="PSP_C"
   operation="recPSP" portType="ns1:recMsg" variable="RecPSPIn">
    <empty name="Empty1"/>
  </onMessage>
  <onMessage partnerLink="PSP_C_int"
   operation="recPSP" portType="ns1:recMoney" variable="SendSPIn1">
    <receive name="recUpdate1" createInstance="no" partnerLink="PSP_C1"
     operation="recPSP" portType="ns1:recMsg" variable="RecPSPIn">
    </receive>
  </onMessage>
  <onMessage partnerLink="PSP_NoC"
   operation="recNoPSP" portType="ns1:recMsg" variable="RecPSPIn">
    <exit name="Exit347"/>
  </onMessage>
</pick>
```

The translation generates the following ISPL program for the client.

```
Agent Client
  Vars:
    state : { Client_0, Client_1, ...};
    count : 0 .. 3;
    ...
  end Vars
  Actions={Client_Upd1_0, Client_Upd1_1,...};
  Protocol :
    state=Client_0:{Client_Upd1_0, Client_Upd1_1,
                    Client_Upd1_2, Client_While1};
    state=Client_1:{Client_Empty1};
    ...
    end Protocol
  Evolution :
    state=Client_0 and count=count+1 if
      state=Client_24 and Action=Client_Assign375;
    ...
  end Evolution
end Agent
```

The following listing gives an example about how to define atomic propositions and properties in ISPL.

```
Evaluation
  Client_green if Client.state = Client_0 or
                  Client.state = Client_1 or ...;
  Client_end if Client.state = Client_51;
  Client_red0 if Client.state = Client_11;
  ...
end Evaluation
Formulae
 E ( Client_green U Client_end );
 EF Client_red0;
  ...
end Formulae
```

In addition to the basic properties automatically generated by the compiler, we manually added a few more complex properties to the model. Those properties were also studied in [1]. Some atomic propositions, e.g., "receiveSoftware" and "softwareTested", are also added to the ISPL code manually. In particular, we considered the following:

- Whenever PSP is in a compliance state, he knows the contract can be eventually fulfilled successfully.

$$AG(PSP\_green \rightarrow K_{PSP}EF(PSP\_end))$$

- There exists a path where C is always in compliance with the contract until he eventually receives the software.

$$E(C\_green\ U\ receiveSoftware)$$

- PSP knows that it is possible that PSP, SP, C, I, H, T and E are all in compliance until the software is delivered.

$$K_{PSP}E(all\_green\ U\ softwareDelivered),$$

  where all_green represents $PSP\_green \wedge SP\_green \wedge C\_green \wedge T\_Green \wedge H\_green \wedge E\_green \wedge I\_green$.

- There is a trace in which the client is always in contract compliant states until the software is delivered (while the client remains compliant) before the client enters a violation.

$$E(C\_green\ U\ E((C\_green \wedge softwareDeployed)\ U\ \neg C\_green))$$

The generated ISPL model was encoded automatically by MCMAS by using 134 BDD variables: 49 BDD variables for local states (the same number of BDD variables are constructed for the transition relation) and 36 for local actions. The total number of global states is approximately 10^6. On a machine running Linux Fedora 8 x86_64 version (kernel 2.6.24.3-50) on Intel Core 2 Duo E4500 2.2GHz with 4GB memory, it took about 24 seconds with 34 MB memory space for MCMAS to verify 25 properties.

In this example, all basic properties hold on the model, which means not only all parties can fulfil their contractual obligations successfully, but also that all the violations shown in Figure 2 can actually happen. Amongst the manually added properties, the first one does not hold. The reason is that even though PSP fulfills its contractual obligations, the software might not pass testing hence not be deployed. For a similar reason, the third one does not hold either.

7 Conclusions

In this paper we presented a novel technique for the verification of contract-regulated service compositions. In our approach, services and contracts are specified as WSBPEL behaviours. We showed how these behaviours could be semi-automatically compiled into ISPL, and then verified using the symbolic model checker MCMAS. The salient feature of the approach is the possibility of checking agent compliance with respect to contracts and the potential of compiling a fairly large subset of BPEL constructs to ISPL. We illustrated the methodology using a realistic case study with a reasonably large state space.

It is worth mentioning that there are two limitations in the current framework: (1) Since MCMAS cannot handle real-time systems, some BPEL constructs such as *deadline* and *timeout* have to be translated into non-deterministic behaviours. For real-time properties, a secondary model checker, such as Uppaal [3] or Verics[6], can be integrated into the framework. (2) The contracts that can be dealt with are written in natural languages and translated into BPEL code manually. Nowadays, some contracting languages, e.g., [14], have been proposed in order to construct electronic contracts to be processed by computers. Currently, we are working on compiling electronic contracts into ISPL to allow more automation.

References

[1] A. Lomuscio and H. Qu and M. Solanki. Towards verifying compliance in agent-based web service compositions. In *Proceedings of The Seventh International Joint Conference on Autonomous Agents and Multi-agent systems (AAMAS-08)*. ACM Press, 2008.

[2] L. Baresi, D. Bianculli, C. Ghezzi, S. Guinea, and P. Spoletini. Validation of web service compositions. *IET Softw.*, 1(6):219–232, December 2007.

[3] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of Uppaal. In *Proceedings of the International Workshop on Software Tools for Technology Transfer*, 1998.

[4] R. Bordini, M. Fisher, C. Pardavila, W. Visser, and M. Wooldridge. Model checking multi-agent programs with CASP. In *CAV'03*, volume LNCS 2725, pages 110–113. Springer-Verlag, 2003.

[5] D Booth, H Haas, F McCabe, E Newcomer, M Champion, C Ferris and D Orchard. Web service architecture. W3c working group note 11 february 2004, 2004. http://www.w3.org/TR/ws-arch/.

[6] P. Dembiński, A. Janowska, P. Janowski, W. Penczek, A. Pólrola, M. Szreter, B. Woźna, and A. Zbrzezny. Verics: A tool for verifying Timed Automata and Estelle specifications. In *Proc. of the 9th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'03)*, volume 2619 of *LNCS*, pages 278–283. Springer-Verlag, 2003.

[7] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Model-based verification of web service compositions. In *Proceedings of the 10th IEEE International Conference on Automated Software Engineering*. IEEE Press, 2003.

[8] X. Fu, T. Bultan, and J. Su. Analysis of interacting BPEL web services. In *13th international conference on World Wide Web*, pages 621–630. ACM Press, 2004.

[9] P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In *Proceedings of 16th International Conference on Computer Aided Verification (CAV'04)*, volume 3114 of *LNCS*, pages 479–483. Springer-Verlag, 2004.

[10] A. Lomuscio, H. Qu, and F. Raimondi. Mcmas 0.9 alpha. `http://sourceforge.net/projects/ist-contract/`, 2008.

[11] A. Lomuscio and F. Raimondi. MCMAS: A model checker for multi-agent systems. In *Proceedings of TACAS 2006*, volume 3920, pages 450–454. Springer Verlag, 2006.

[12] A. Lomuscio and M. Sergot. Deontic interpreted systems. *Studia Logica*, 75(1):63–92, 2003.

[13] OASIS Web service Business Process Execution Language (WSBPEL) TC. Web service Business Process Execution Language Version 2.0, 2007.

[14] S. Panagiotidi, J. Vazquez-Salceda, S. Alvarez-Napagao, S. Ortega-Martorell, S. Willmott, and P. Storms R. Confalonieri. Contracting agent language. In *Symposium on Behaviour Regulation in Multi-Agent Systems*, 2008.

[15] W. Penczek and A. Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. *Fundamenta Informaticae*, 55(2):167–185, 2003.

[16] Marco Pistore, F. Barbon, Piergiorgio Bertoli, D. Shaparau, and Paolo Traverso. Planning and monitoring web service composition. In *AIMSA*, pages 106–115, 2004.

[17] M. Wooldridge. *An introduction to multi-agent systems*. John Wiley, England, 2002.

[18] X. Fu T. Bultan and J. Su. Conversation Protocols: A Formalism for Specification and Verification of Reactive Electronic Services. In *CIAA*, volume LNCS 2759, pages 188–200. Springer-Verlag, 2003.

Security-By-Contract for the Future Internet *

Fabio Massacci[1], Frank Piessens[2], and Ida Siahaan[1]

[1] Universita' di Trento, Italy name.surname@disi.unitn.it
[2] Katholieke Universiteit Leuven, Belgium name.surname@cs.kuleuven.be

1 The Future Internet

With the advent of the next generation java servlet on the smartcard, the Future Internet will be composed by web servers and clients silently yet busily running on high end smart cards in our phones and our wallets. Thus we can no longer accept the current security model where programs can be downloaded on our machines just because they are vaguely "trusted". We need to know what they do in more precise details.

The End of Trust in the Web. The World Wide Web evolved rapidly in 90's and the notion has changed from a network to a platform where people migrate desktop applications. The security model of the current version of the web is based on an assumption that the good guys develop their application, expose it on the web, and then let other good guys using it while stopping bad guys from misusing it.

The business trend of outsourcing processes or the construction of virtual organizations have slightly complicated this initially simple picture. Now running a "service" means that different service (sub)components can be dynamically chosen and different partners are chosen to offer those (sub)services. Hence we need different trust establishment mechanisms (see e.g. [10]).

This assumption is no longer true for the new world of Web 2.0 and the Future Internet. Even now a user downloads a multitude of communicating applications ranging from P2P clients to desktop search engines, each of them ploughing through the user's platform, and springing back with services from and to the rest of the world. To deal with the untrusted code either .NET or Java can exploit the mechanism of permissions. Permissions are assigned to enable execution of potentially dangerous or costly functionality, such as starting various types of connections. The drawback of permissions is that after assigning a permission the user has very limited control over how the permission is used. Conditional permissions that allow and forbid use of the functionality depending on such factors as bandwidth or the previous actions of the application itself (e.g. access to sensitive files) are also out of reach. Once again the consequence is that either applications are sandboxed (and thus can do almost nothing), or the user decided that they are trusted and then they can do almost everything.

The mechanism of signed assemblies from trusted third parties does not solve the problem either. Currently a signature on a piece of code only means that the application

* Research partly supported by the Projects EU-FP6-IST-STREP-S3MS, EU-FP6-IP-SENSORIA, and EU-FP7-IP-MASTER. We would like to thank Eric Vetillard for pointing to us the domain of Next Generation Java Card as the Challenge for the Future Internet.

comes from the software factory of the signatory, but there is no clear definition of what guarantees it offers. It essentially binds the software with nothing. We built our security models on the assumption that we could trust the vendors (or at least some of them). The examples from reputable companies such as Channel 4 (or BBC, Sky TV etc.) show that this is no longer possible. Still we really want to download a lot of software.

The Smart(Card) Future of the Web. The model that we have described above is essentially the web of the personal computers. None of the users complaining about 4oD [14] have considered their PC or their Web platform "broken" because it allowed other people to make use of it. They did not consider returning their PC for repair. They considered themselves being gullible users ripped off by an untrusted vendor.

Another domain at the opposite side of the psychological spectrum is smartcard technology. The technology enjoyed worldwide deployment in 90's with Java Card Applets and their strict security confinement. At the beginning of the millennium, many applications such as large SIM cards and identity management businesses are implemented on smart-cards to address mobile devices security challenges.

The smartcard technology evolved with larger memories, USB and TCP/IP support and the development of the Next-Generation Java Card platform with Servlet engine. The Future Internet will be composed by those embedded Java Card Platforms running on high end smart cards in our phones and our wallets, each of them connecting to the internet and performing secure transactions with distributed servers and desktop browsers without complicated middleware or special purpose readers.

We still want to download a huge amount of software on our phones but there is a huge psychological difference from a consumer perspective. If our PC is sluggish in responding, *we* did something wrong or downloaded the wrong software, if our phone is sluggish, *it* is broken. Moreover, in the realm of next generation Java card platforms we cannot just download a software without knowing what it does. The smart card web platform must have a way to check what is downloading.

2 Security by Contract for the Smart Future Internet

In the previous FLACOS workshop we [11] have proposed the notion of Security-by-Contract (S×C)[5, 4]. In S×C we augment mobile code with a claim on its security behavior (an *application's contract*) that could be matched against a mobile *platform's policy* before downloading the code. A digital signature does not just certify the origin of the code but also bind together the code with a contract with the main goal to provide a semantics for digital signatures on mobile code. This framework is a step in the transition from trusted code to trustworthy code.

S×C *Workflow.* At *development time* the mobile code developers are responsible for providing a description of the security behavior that their code finally provides. Such a code might also undergo a formal certification process by the developer's own company, the smart card provider, a mobile phone operator, or any other third party for which the application has been developed. By using suitable techniques such as static analysis, monitor in-lining, or general theorem proving, the code is certified to comply with the

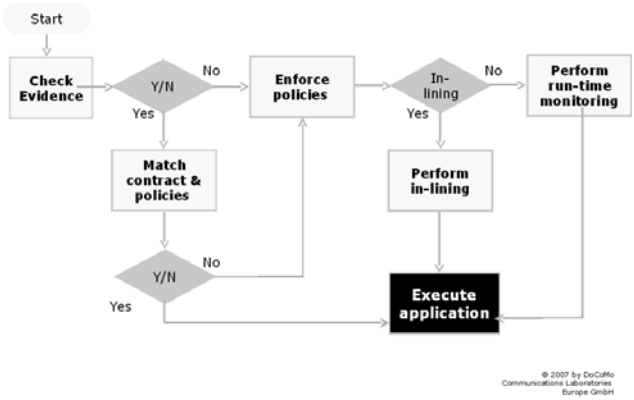


Fig. 1. SxC Workflow

developer's contract. Subsequently, the code and the security claims are sealed together with the evidence for compliance (either a digital signature or a proof) and shipped for deployment. At *deployment time*, the target platform follows a workflow similar to the one depicted in Fig.1 (see [19]). First, it checks that the evidence is correct. Such evidence can be a trusted signature or a proof that the code satisfies the contract (one can use Proof-Carrying-Code (PCC) techniques to check it.

As we have evidence that the contract is trustworthy, the platform checks, that the claimed policy is compliant with the policy that our platform wants to enforce. If it is, then the application can be run without further ado. It is a significant saving from in-lining a security monitor. In case that at *run-time* we decide to still monitor the application then, as with vaccination, we inline a number of checks into the application so that any undesired behavior can be immediately stopped or corrected.

Contract for the Smart Future Internet. A *contract* is a formal complete and correct specification of the behavior of an application for what concerns relevant security actions (Virtual Machine API Calls,Web Messages etc). By signing the code the developer certifies that the code complies with the stated claims on its security-relevant behavior. A *policy* is a formal complete specification of the acceptable behavior of applications to be executed on the platform for what concerns relevant security actions.

Technically, a contract can be a security automaton in the sense of Schneider [8], and it specifies an upper bound on the security-relevant behavior of the application: the sequences of security-relevant events that an application can generate are all in the language accepted by the security automaton. We can have a slightly more sophisticated approach using Büchi automata [18] if we also want to cover liveness properties that can be enforced by Edit automata. This definition can be sufficient for theoretical purposes but it is hardly acceptable for any practical use.

A variant of the PSLANG language [1] has been proposed for S×C for mobile code (.NET and Java). The formal counterpart of the language is the notion of *automata modulo theory* [12] where atomic actions are replaced by expressions that can finitely

capture infinite values of API parameters. For the smart future internet, we need to identify a suitable language for the specification of contracts and policies at a level of abstraction that is suitable and can be used for *all* S×C phases (Fig.1)

Application-contract compliance. Static analysis can be used at development time to increase confidence in the contract. With static analysis, program analysis and verification algorithms are used in an attempt to *prove* that the application satisfies its contract.

The major advantage of static analysis is that it does not impose any runtime overhead, and that it shows that all possible executions of a program comply with the contract. The major disadvantage is that the problem of checking application-contract compliance is in general undecidable, and so automatic static analysis tools will typically only support restricted forms of contracts, or restricted forms of applications, or the tool will be *conservative* in the sense that it will reject applications that are actually compliant, but the tool fails to find a proof for this.

The programs and services running on the embedded servlet will be significantly more complex and have actions at different level of abstractions whose full security implications can be understood by considering all abstraction levels at once. The challenges for static analysis is that with expressive notions of security contracts, verifying application-contract compliance is actually as hard as verifying compliance with an arbitrary specification [16]. Moreover, contracts for applications in the Smart Future Internet will have a complexity that is comparable to the level of abstractions of current concurrent models that are used for model checking hardware and software systems (in 10^{10} states or transitions and beyond).

A standard approach to make program verification and analysis algorithms scale to large programs is to make them *modular* of the program independently. This is particularly hard for application-contract compliance checking, because the security state of the contract is typically a global state, and the structure of the contract and its security state might not align with the structure of the application. Annotations are required on *all* methods to specify how they interact with the security state, and not only on methods that are relevant for the contract at hand. This annotation overhead is prohibitive, so a key challenge is to look for ways to reduce the annotation burden. An interesting research question is whether a program transformation (similar to the security-passing style transformation used for reasoning about programs sandboxed by stack inspection [17]) can improve this situation.

A second approach to address scalability is to give up soundness of the analysis, and to use the contract as a model of the application in order to generate security tests by applying techniques from Model Based Testing [20]. Losing soundness is a major disadvantage: an application may pass all the generated tests and still turn out to violate the contract once fielded. However, the advantages are also important: no annotations on the application source code are needed, and the tests generated from the contract can be easily injected in the standard platform testing phase, thus making this approach very practical. A challenge to be addressed here is how to measure the coverage of such security tests. When are there enough tests to give a reasonable assurance about security? It is easy to automatically generate a huge amount of tests from the contract. Hence it is important to know how many tests are sufficient, and whether a newly generated test increases the coverage of the testing suite.

Matching Contract and Policy on the Smart Future Internet. We must show that the behavior described by the contract is acceptable according to our platform policy. The operation of matching the application's claim with the platform policy requires that the contract is trustworthy, i.e. the application and the contract are sealed together with a digital signature when shipped for deployment or by shipping a proof that can checked automatically. A simple solution is to build upon automata theory, interpret contract and policy as automata and use language inclusion . Given two such automata Aut^C (representing the contract) and Aut^P (representing the policy), we have a match when the language accepted by Aut^C is a subset of the language accepted by Aut^P.

Once the policy and the contract are represented as automata then one can either use language inclusions [12] or simulation [13] to check whether the contract is acceptable according our platform policy. This solution is only partial because the automata that we have envisaged do not store the values of the arguments of allowed/disallowed APIs. In order to do this Contracts and policies for the future internet must be history-dependent: the arguments of past allowed actions (API calls, WS invocations, SOAP messages) may influence the evolution of future access control decision in a policy.

Further, in our current implementation of the matcher that runs on a mobile phone, security states of the automata are represented by variables over finite domains e.g. smsMessagesSent ranges between 0 to 5. [1, 2]. A possible solution could be to extend the work on finite-memory automata [9] by Kaminski and Francez or other works [15] that studied automata and logics on strings over infinite alphabets.

An approach to address scalability is to give up soundness of the matching and use algorithms for simulation and testing. A challenge to be addressed is how to measure the coverage of approximate matching. Which value should give a reasonable assurance about security? Should it be an absolute value? Should it be in proportion of the number of possible executions? In proportion to the likely executions? An interesting approach could be to recall to life a neglected section on model checking by Courcoubetis et al [3] in which they traded off a better performance of the algorithm in change for the possibility of erring with a small probability.

Inlining a monitor on Future Internet Applications. What happens if matching fails? or what happens if we do not trust the evidence that the code satisfies the contract? If we look back at Fig.1 monitor inlining of the *contract* can provide strong assurance of compliance. With *monitor inlining* [7], code rewriting is used to push contract checking functionality into the program itself. The intention is that the inserted code enforces compliance with the contract, and otherwise interferes with the execution of the target program as little as possible. Monitor inlining is a well-established and efficient approach [6] however a major open question is how to deal with concurrency efficiently.

Servents in the Smart Future Internet will need to monitor the concurrent interactions of tens of untrusted multithreaded programs. An inliner needs to protect the inlined security state against race conditions. So all accesses to the security state will happen under a lock. A key design choice for an inlining algorithm is whether to lock across security relevant API calls, or to release the lock before doing the API call, and reacquiring it when the API call returns.

The first choice (locking across calls) is easier to get secure, as there is a strong guarantee that the updates to the security state happen in the correct order. This is much

trickier for an inliner that releases the lock during API calls. However, an inliner that locks across calls can introduce deadlocks in the inlined program, because some of the security relevant API calls will themselves block. And even if it does not lead to deadlock, acquiring a lock across a potentially blocking method call can cause serious performance penalties. A partial solution is by partitioning the security state into disjoint parts, and replacing the global lock, by per-part locks. This improves efficiency, but depending on application and policy, it can still introduce deadlocks. The challenge is how to inline a monitor into a concurrent program so that it cannot create a deadlock in future interactions with other unknown programs yet to be downloaded.

The ability to resist to changes in context (i.e. new concurrent programs downloaded after the inlined program) is essential for usability. The inlined version of 4oD should not get in the way if later on I want to download a (inlined) role-playing game. It is possible that two malicious software downloaded at different instants try to cooperate in order to steal some data. The security monitor should be able to spot them but not be deadlocked by them. If inlining is performed by the code producer, or by a third party, the code consumer (client that runs the application) needs to be convinced that inlining has been performed correctly. Without a secure transfer of the guarantees of application-contract compliance to the client, it is easy for an attacker to modify either the application or the contract, or for an application developer to lie about the contract.

Cryptographic signatures by a trusted (third) party is a first solution even if it transfer the risk from the technical to the legal domain. The trusted party vouches for application-contract compliance. Note the difference with the use of signatures in the traditional mobile device security model. In the security-by-contract approach, a signature has a clear semantics [5]: the third party claims that the application respects the supplied contract. Moreover, what is important is the fact that the decision whether the contract is acceptable or not remains with the end user. If an application claims that it will not connect to the internet and instead it does, at least you can bring the signatory to the court for fraudulent commercial claims.

Another solution is whether we can use the techniques PCC for this. In PCC, the code producer produces a proof that the code has certain properties, and ships this proof together with the code to the client. By verifying the proof, the client can be sure that the code indeed has the properties that it claims to have.

The difficulty of the endeavour is that the code has not been produced to be verified compliant against a security property but usually to actually do some business. In other words, the code producer is not aware of the property and the property producer is not aware of the code. In this scenario verification is clearly an uphill path.

When we inline a contract we know precisely what code we are inlining and also what property the inlined code should satisfy. So, we can ask the inliner to do this automatically for us and ask them to generate the proof directly. This should make it relatively easy to check that code complies with the contract: the generation of a proof should be easier, and the size of the proof would also be acceptable for inlined programs. The challenge is to identify automatic inlining mechanisms that inline a monitor for a security contract and generate an easily checkable proof for industrial applications in the Smart Future Internet.

References

1. I. Aktug and K. Naliuka. Conspec - a formal language for policy specification. Proc. of the 1st Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM2007), 2007.
2. N. Bielova, M. Dalla Torre, N. Dragoni, and I. Siahaan. Matching policies with security claims of mobile applications. In *Proc. of the 3rd Int. Conf. on Availability, Reliability and Security (ARES'08)*. IEEE Press, 2008.
3. C. Courcoubetis, M.Y. Vardi, P. Wolper, and M. Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal Methods in Sys. Design*, 1(2-3):275–288, 1992.
4. L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe. Security-by-contract on the .net platform. *Elsevier Inform. Sec. Technical Report*, 13(1):25–32, 2008.
5. N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-Contract: Toward a Semantics for Digital Signatures on Mobile Code. In *Proc. of the 4th European PKI Workshop Theory and Practice (EUROPKI'07)*. Springer-Verlag, 2007.
6. U. Erlingsson and F.B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. of the 1999 New Security Paradigms Workshop (NSPW'99)*.
7. U. Erlingsson and F.B. Schneider. IRM enforcement of Java stack inspection. In *Proc. of the 2000 IEEE Symp. on Security and Privacy*, pages 246–255. IEEE Computer Society, 2000.
8. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *TOPLAS*, 28(1):175–205, 2006.
9. M. Kaminski and N. Francez. Finite-memory automata. *Theor. al Comp. Sci.*, 134(2):329–363, 1994.
10. Y. Karabulut, F. Kerschbaum, F. Massacci, P. Robinson, and A. Yautsiukhin. Security and trust in it business outsourcing: a manifesto. In S. Etalle and P Samarati, editors, *Proc. of the 2nd Int. Workshop on Security and Trust Management (STM'06)*, ENTCS. Elsevier, 2006.
11. F. Massacci, N. Dragoni, and I. Siahaan. A Security-by-Contracts Architecture for Pervasive Services. 2007.
12. F. Massacci and I. Siahaan. Matching midlet's security claims with a platform security policy using automata modulo theory. In *Proc. of The 12th Nordic Workshop on Secure IT Systems (NordSec'07)*, 2007.
13. F. Massacci and I. Siahaan. Simulating midlet's security claims with automata modulo theory. In *Proc. of the 2008 workshop on Prog. Lang. and analysis for security*, 2008. submitted.
14. CNET Networks. Channel 4's 4od: Tv on demand, at a price. *Crave Webzine*, January 2007.
15. F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *TOCL*, 5(3):403–435, 2004.
16. F.B. Schneider. Enforceable security policies. *ACM Trans. on Inf. and Sys. Security*, 3(1):30–50, 2000.
17. J. Smans, B. Jacobs, and F. Piessens. Static verification of code access security policy compliance of .net applications. *J. of Object Technology*, 5(3):35–58, 2006.
18. C. Talhi, N. Tawbi, and M. Debbabi. Execution monitoring enforcement under memory-limitation constraints. *Inform. and Comp.*, 206(2-4):158–184, 2007.
19. D. Vanoverberghe, P. Philippaerts, L. Desmet, W. Joosen, F. Piessens, K. Naliuka, and F. Massacci. A flexible security architecture to support third-party applications on mobile devices. In *Proc. of the 1st ACM Comp. Sec. Arch. Workshop*, 2007.
20. M. Veanes, C. Campbell, W. Schulte, and N. Tillmann. Online testing with model programs. In *Proc. of the 10th Eur. Software Eng. Conf. held jointly with 13th ACM SIGSOFT Int. Symp. on Found. of Software Eng.*, pages 273–282. ACM Press, 2005.

Increasing Trust in Public Service Delivery – Contract-Based Software Infrastructure for Electronic Government

Adegboyega Ojo and Tomasz Janowski

Center for Electronic Governance
United Nations University IIST
P.O. Box 3058, Macao
{ao,tj}@iist.unu.edu

ABSTRACT: Citizens increasingly demand high-quality public services, available from a one-stop government portal, and delivered through a choice of electronic and traditional channels. In order to fulfill this demand, the implementation involves collaboration between agencies at different levels of government, driven by complex administrative and business processes. In addition, the delivery of public services increasingly involves private sector organizations serving as intermediaries or suppliers, able to flexibly join or leave dynamic service networks. Such networks have to be managed to address the expected stability of the services delivered through them.

This paper presents a lightweight software infrastructure that enables collaborative delivery of public services through a managed set of services, components and frameworks. The infrastructure currently comprises five run-time and design-time elements: (1) Front Office Framework, (2) Back Office Framework, (3) Workflow Service, (4) Messaging Service, and (5) Infrastructure Management Service. With each element offering a pair of functional and management interfaces, the infrastructure allows the expression, instrumentation and verification of contracts. Contracts are expressed in an XML language defined by us, their instrumentation is implemented through the Web Services Distributed Management (WSDM) framework, and verification is carried out though a mediation service, part of Management Service. In addition to infrastructure-level contracts, we briefly discuss how the effects of infrastructure level contracts on high-level service agreements at process and service levels. Finally, we comment on how contract management at infrastructure level can be integrated into the IT Governance and specifically, Service Management practices, towards a trustworthy public services delivery system.

1. Introduction

The delivery of public services is one of major functions of any government. Most public administration modernization programs are targeted at transforming traditional service delivery to seamless, multi-channel and highly personalized services through citizen relationship management approaches. These modernization efforts are aimed at efficiency gains, cost savings and most importantly, at increasing citizen trust in government.

Several studies have shown that citizen trust depends on the factors such as government commitment to information protection, third-party assurance, privacy policy, accessibility, security policy, past experience and others [1]. While citizen trust is behavioral, it is indirectly related to the trustworthiness properties of ICT systems supporting government institutions [2], provided that organizational and process aspects of the citizen-government interaction is managed appropriately. However, no matter how well citizen relationship is managed by government, gaining citizen trust is impossible without relying on trustworthy ICT-infrastructure supporting service delivery and government-citizen

interactions in general. Therefore, citizen trust can be improved by guaranteeing in particular the trustworthiness of the government ICT infrastructure in terms of major citizen trust factors, such as privacy and security. In [3], Gates defined the notion of Trustworthy Computing to be computing that is as available, reliable and secure, as electricity, water services and telephony.

Establishing trustworthiness of electronic public services and their underlying software infrastructure to support service pledges made by government agencies to their citizens can be challenging for many reasons. One of them is the complex nature of the e-government infrastructure, typically consisting of services (e.g. authentication, time-stamping, notification, workflow and messaging) executed in a heterogeneous distributed computing environment, across different organizations, with different levels of service management (or IT governance). In addition, services requested by citizens are often implemented through business processes spanning the boundaries of various agencies and involving organizations outside government, for instance private sector and community based organizations.

In this ongoing work, we describe how Quality of Service (QoS) contracts can be used as basis for managing electronic public service infrastructure to realize a trustworthy service delivery environment in government. In particular, we show how QoS contracts can be specified, monitored and mediated. Contract specification is expressed in a simple XML language. The language allows users to specify identifiers for provider and client services (or applications) and the agreed settings for a series of QoS parameters. Monitoring is realized through web service intermediaries (service handlers or proxies) implementing the OASIS Web Services Distributed Management specifications. Verification consists of processing QoS contracts defined between pairs of services and the QoS-related information in the service provider's operation records. Mediation is effected to execute specific actions on the offending party, usually the service provider. The described QoS contract management capability is implemented as an Infrastructure Management system integrated with other elements of a software infrastructure for Electronic Government, developed by the Center for Electronic Governance at UNU-IIST.

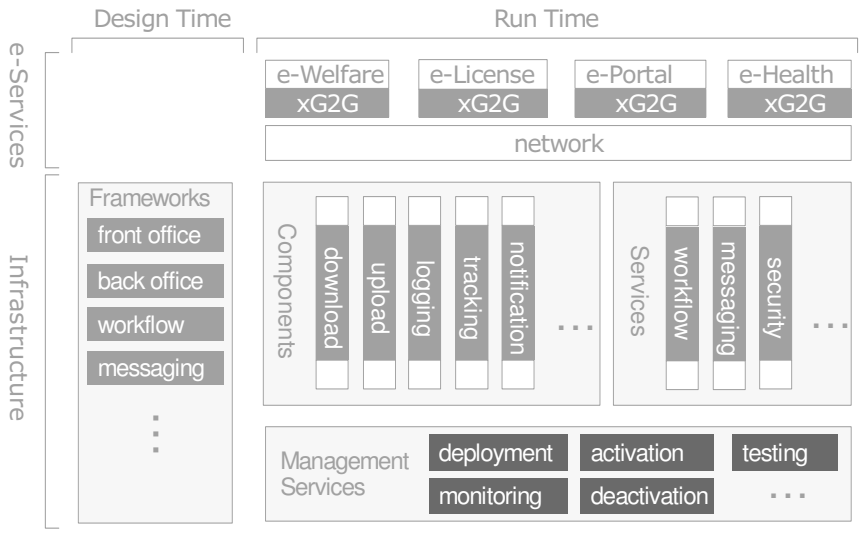
2. Electronic Public Service (EPS) Infrastructure

Large-scale development and deployment of electronic public services requires a significant investment in the underlying software infrastructure, given that such an infrastructure must support the rapid development and deployment of such services. Most e-service infrastructures provide the services that implement basic features and reusable business patterns, such as: security and authentication, user profiling, e-payment, auditing and notification services, address verification and acknowledgement. Specifically, the EPS infrastructure developed by the Center consists of two categories of elements [4]:

- Design-time elements used to develop EPS and
- Run-time elements used to provide the functions required by EPS.

The former consist of the frameworks for developing Front-Office and Back-Office parts of an EPS [5, 6]. The latter consist of Workflow and Messaging services [4, 7], as well as components for: downloading, uploading, logging, tracking and other generic functions. To manage both design and runtime elements of the infrastructure, a Management Service was implemented as part of the EPS infrastructure which administers the entire life-cycle of infrastructure elements: from registration and startup, through monitoring and control, to shutdown. Figure 1 provides a schematic view of the EPS infrastructure [8].

Figure 1: Schema of EPS Infrastructure



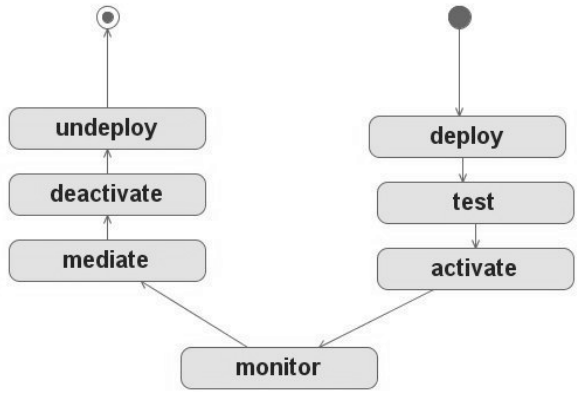
We present in subsequent sections the details on how the Management Service of the EPS Infrastructure administers other elements of the infrastructure through QoS contracts.

3. EPS Infrastructure Management through Contracts

3.1. Management Requirements

All EPS Infrastructure elements offer two sets of interfaces: (1) functional interface for specialized operations and (2) management interfaces for manageability. In addition, for administrative purposes, non-service elements are wrapped as web services. Thus, the initial set of management requirements were guided by the W3C web service lifecycle definition shown below.

Figure 2: Management Lifecycle of Infrastructure Elements



The requirements are as follows:

Requirement	Description
Deployment	Registration and setting up of an element on the infrastructure
Testing	Ensuring that the element executes correctly on the infrastructure
Activation	Starting up registered elements on the infrastructure
Monitoring	Collecting operational data on the element at run-time
Mediation	Controlling the behavior of elements based on available operational data
Deactivation	Stopping the execution of services and components
Undeployment	Un-registering components or applications from the infrastructure

In addition to these requirements, the management service provides registration and authentication for client services and applications, as well as contract management between infrastructure elements and client applications.

By wrapping infrastructure elements as web services, we enable the management of the whole infrastructure through the OASIS Web Service Distributed Management Framework (WSDM) [9] and its sub-specifications – Management Using Web Services (MUWS)[10] and the Management using Web Services (MOWS)[11]. In the following, we discuss how the notion of contract is integrated with the WSDM for effective management of the EPS infrastructure.

3.2. Management by Contract

A contract establishes a formal relationship between two or more parties that use or provide resources, where rights, obligations and negotiation rules over the used resources are expressed [14]. In the context of the EPS infrastructure, a contract formally relates any infrastructure element with other infrastructure elements or external services in the role of service providers and consumers. Quality of Service (QoS) refers to non-functional aspects of a service such as performance, reliability, availability, security, which are associated with the specialized functions provided by the service.

QoS contracts define the QoS parameters agreed upon between service consumers and providers. Obligations define the guarantees, constraints, and penalties based on the actual measured QoS parameters, and those specified in the QoS contracts. In the case of the EPS Infrastructure, IMS collects and stores operational data, including QoS data through its monitoring function.

3.3. QoS Contract Specification

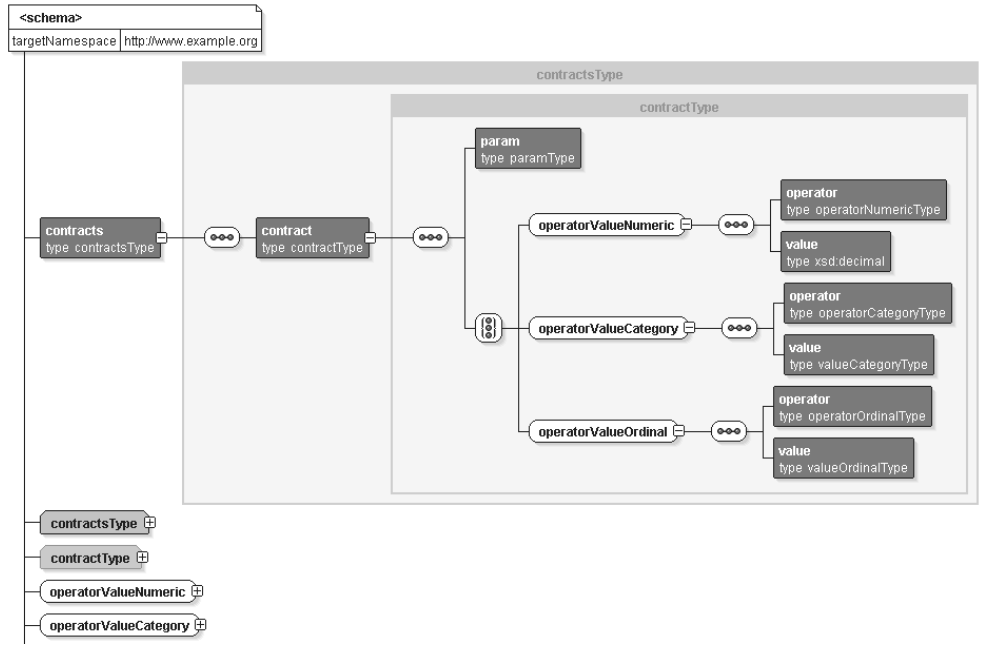
A number of languages have been proposed for specifying QoS contracts, for instance the WSLA by IBM [18] and QML [15]. In addition, a comprehensive QoS contract management framework was presented in [15], which also provides an initial generic set of QoS parameters based on three categories of services: task-oriented services, message-oriented services and streaming media.

The first two set of services are applicable to the EPS infrastructure since there are specialized services for workflow management, notification, tracking (task-oriented), in addition to the messaging service. Important generic QoS characteristics cataloged in [15] for these two categories of services include: accountability, availability, confidentiality, criticality, deadline, information accuracy, information

throughput, integrity, message delay, message delivery guarantee or loss ratio, message duplication elimination, message ordering and retry limits.

Based on these parameters, we define a simple XML–based language for specifying QoS contracts. The language represents a QoS contract as a set of tuples consisting of a QoS parameter, an operator and an associated value. A QoS parameter may be numeric (integer or real numbers), ordinal (Low, Medium, High) or categorical (e.g. True and False). See Figure 3 below for a diagram.

Figure 3: QoS Contract Specification Language



For management purpose, each contract is associated with a pair of entities representing the service provider and the service consumer.

3.3. Contract Monitoring and Verification

Monitoring is achieved by intercepting request to Infrastructure elements and logging the relevant operational information on the involved elements. For logging, the MUWS and MOWS definitions (properties) for web service resources were extended to include properties defined by the QoS Specification language in the previous Section.

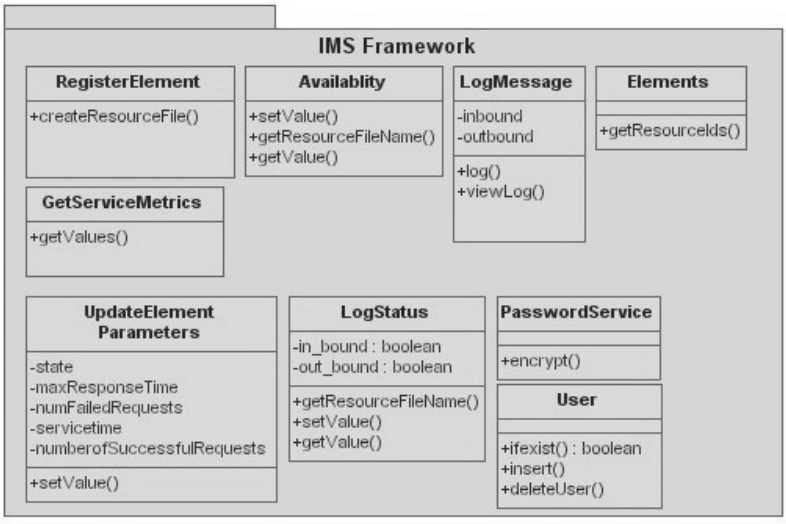
For verification, specified contracts are checked periodically against actual operational data logged through the monitoring function to determine non-compliance. In exceptional cases, reports are generated and sent to the mediation service for remediation or action.

The following section describes the prototype implementation of the EPS Infrastructure Management.

4. Infrastructure Management System (IMS) – Prototype Implementation

The IMS system consists of a web application (client for human user), a web tier sub-system and a framework providing APIs for supported management functions including – registration of infrastructure elements and clients checking availability of services, logging (monitoring), retrieving service metrics (for verification), updating operational data (monitoring) and authentication. Each infrastructure element is associated with a unique Resource Property Document created during the registration of the element on the infrastructure.

Figure 4: IMS Framework APIs

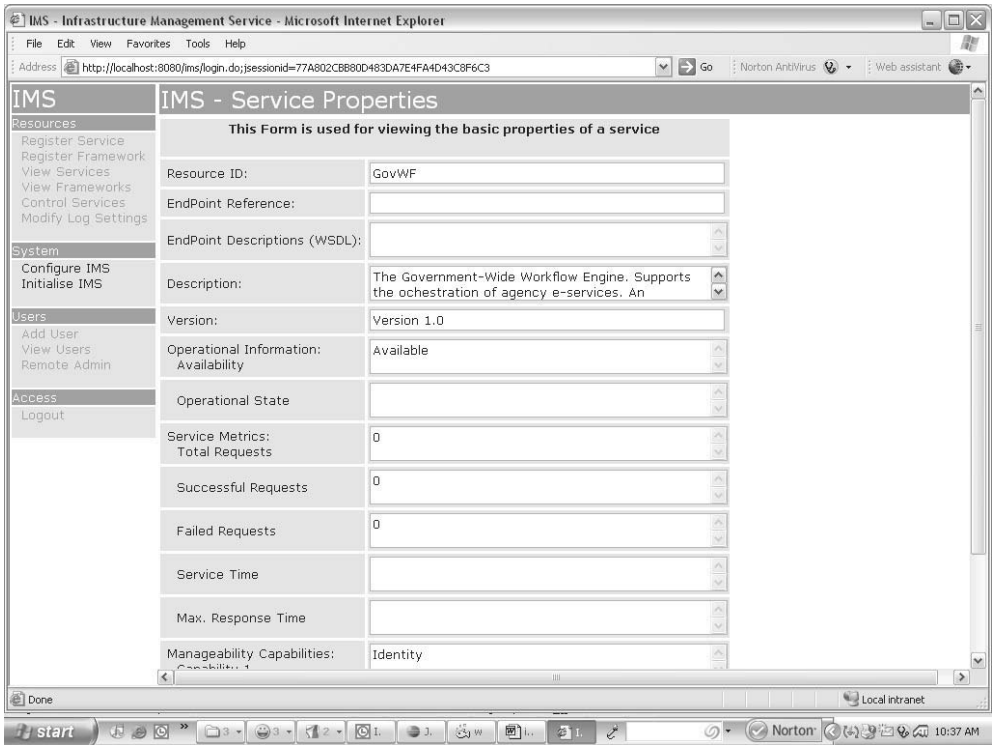


The structure of the Resource Property Document is based on the MUWS [10] and MOWS [11] specification but adapted to suit the requirements for the EPS Infrastructure management, particularly in the aspect of QoS contract specification and verification. The framework APIs provide useful functions for implementing basic lifecycle and user management functions through the operations on the Resource Properties Documents.

The IMS Application is deployed as a typical web application on the Apache Tomcat Application Server. All infrastructure elements to be managed are exposed as web services through the IMS Handler. The Handler is a standard web service intermediary that must be deployed within the SOAP engine used by the service to be managed.

Figure 5 shows a screenshot from the IMS application showing how services are registered on the infrastructure. The technical detail of the IMS component of the EPS Infrastructure is documented in 8.

Figure 5: Screenshot for Viewing Service Properties



5. Conclusions

This paper describes a work in progress on the design and implementation of a contract-based EPS infrastructure. Infrastructure elements provide specialized services and also a set of management services constituting the Infrastructure Management Service. The contract specification language and the mechanisms for monitoring, verification and mediation of contracts were briefly discussed. We also briefly discussed the prototype implementation of the EPS Infrastructure Management Service.

Our ultimate goal in this work is to develop a comprehensive contract-based framework which will establish contract hierarchies (or dependencies) by linking contracts specified at modules and functions (based on preconditions, post conditions and invariants) to contracts at the infrastructure level (as discussed in this paper) and finally to contract for business processes in a way to provide early warning on failures at infrastructural level and ultimately at service levels (on performance pledges to citizens).

Organizationally, processes would be developed based on the framework and integrated into service management and IT governance practices; government-wide towards building a trustworthy electronic service delivery environment and earning citizen's trust.

References

[1] Seung-Yong and Lung-Teng Hu, Citizen's trust in Digital Government, Graduate Department pf Pubic Admin, Rutgers, the State University of New Jersey.

[2] Chrisanthi Avgerou, Andrea Ganzaroli, Angeliki Poulymenakou, Nicolau Reinhard, ICT and Citizen's trust in government: lessons from electronic voting in Brazil, Proceedings of the 9th Conference on Social Implications of Computers in Developing Countries, Sao Paolo, 2007

[3] Bill Gates: Trustworthy Computing, [online] available at http://www.wired.com/techbiz /media/news/2002/01/49826

[4] Ojo, A., Janowski, T., Oteniya, G., Estevez, E., Infrastructure Support for e-Government – An Overview, e-Macao Report 6, Aug. 2006.

[5] Ojo, A., Chu, T. I., Oteniya, G., Tou, C. P., Janowski, T., Front-Office Framework for e-Government, e-Macao Report 8, Aug. 2006.

[6] Estevez, E., Wan, C., Wong, C. T., Ojo, A., Oteniya, G., and Janowski, T., Back Office Framework for e-Government, e-Macao Report 9, Aug. 2006

[7] Estevez, E., and Janowski, T., Extensible Message Gateway for e-Government, e-Macao Report 11, Aug. 2006.

[8] Ojo, A., Oteniya, G., and Janowski, T., Infrastructure Management Services for e-Government, e-Macao Project Report 12, Aug. 2006.

[9] An Introduction to WSDM, Committee Draft, OASIS, Available online http://www.oasis-open.org/committees/download.php/16998/wsdm-1.0-intro-primer-cd-01.doc, Feb. 2006.

[10] Web Services Distributed Management: MUWS Primer, Committee Draft, OASIS, Available online http://www.oasis-open.org/committees/download.php/17000/wsdm-1.0-muws-primer-cd-01.doc, February 24, 2006

[11] Web Services Distributed Management: MOWS Primer, Committee Draft, OASIS, Available online http://www.oasis-open.org/committees/download.php/17001/wsdm-1.0-mows-primer-cd-01.doc, February 24, 2006.

[12] Web Services Management: Service Life Cycle, W3C Working Group Note, Available online at http://www.w3.org/TR/wslc/11 February 2004.

[13] Struts in Action, Ted Husted, Cedric Dumoulin, George Franciscus, David Winterfeldt, Manning Greenwich, 2003.

[14] Orlando Loques and Alexandre Sztajnberg, Customizing Component-Based Architectures by Contract, Component Deployment, pp 18-34, 2004.Svend Frølund and Jari Koistinen, Quality of services specification in distributed object systems design. In Proceedings of the 4th Conference on USENIX Conference on Object-Oriented Technologies and Systems - Volume 4 (Santa Fe, New Mexico, April 27 - 30, 1998). Conference on Object-Oriented Technology and Systems. USENIX Association, Berkeley, CA, 1-1, 1998.

[15] Changzhou Wang, Guijun Wang, Haiqin Wang, Alice Chen, and Rodolfo Santiago, Quality of Service Contract Specification, Establishment, and Monitoring for Service Level Management, Vol. 6, No. 11, Special Issue on Advances in Quality of Service Management, December 2007.

[16] IBM, Web Service Level Agreements (WSLA) Project, http://www.research.ibm.com/wsla/

Service Oriented Architectures: The new software paradigm

W. Reisig
Humboldt-Universität zu Berlin

Abstract:

Service oriented architectures are expected to become the foundational layer for tomorrow's information systems and are influencing already many application areas. The principles of SOA have evolved along applications, but the goals of SOA are far more ambitious, requiring a decent formal, abstract basis, and in particular adequate modelling techniques. This paper surveys some problems, discusses some elements of an abstract basis of SOA and outlines a generic modelling technique.

1. Introduction

The potential of *service oriented architectures (SOA)* is widely acknowledged. Recent voices from industrial labs praise SOA as "THE most relevant emerging software paradigm", "a substantial change of view, as it happens at most once each decade", "the next fundamental software revolution after OO" or "much more than just another type of software". These architectures are expected to become the foundational layer for tomorrow's information systems and are influencing already many application areas like Enterprise Application Integration, Software Engineering, Systems Management, Data provisioning, BPI, B2B – to name but just a few. SOAs support the quick and efficient coupling of encapsulated software components ("services"), as well as flexibility and convenience of interaction. Historically, SOA has emerged from two very pragmatic sources and backgrounds: business process technology and Web service technology. So far, SOA is frequently conceived as a means to equip business processes with web-based communication facilities.

The goals of SOA are however far more ambitious: SOA is the next step in trying to bring some order (modularization, proper interfaces, standardization) to enterprise computing and enterprise application integration. In these areas, the state of the art is at the level of spaghetti code. SOA is, in fact, the first serious attempt at bringing some structure into that world. SOA can be done without web services and without business processes.

Adequate representation, communication, and analysis of service-oriented architectures require specific *modelling* techniques. Many techniques and languages originally focussing on business processes are increasingly applied to SOA, too. The most prominent ones are BPMN, EPC (event-process chains), YAWL, ADEPT, and versions of UML (activity) diagrams. Moreover, dedicated implementation oriented languages such as WS-BPEL are used for human communication about services. The most promising recent approach is the Business *Process Modelling Notation* (BPMN), supported by leading companies of the software industry. Languages such as BPMN support communication about (cross-organizational) business processes. Each modelling language comes with a tool or a tool set, supporting graphical representation, translation to executable software, and – to a minor extent – correctness and consistency tests. However, those techniques and languages appear not as attractive and are not as widely used as they could and should be. Not only the software industry, but also textbooks describe SOA usually by means of plain English, informal graphics, pseudo code, and programs in concrete programming languages. But those means are too often ambiguous, and concrete programs tend to mix substantial aspects with coincidental ones.

In the rest of this paper we suggest that SOAs can be constructed cheaper, quicker, better understandable, more reliable, simpler maintainable, etc, by means of *models*. This applies in particular to the following problems:

Service Composition: In a common scenario, established business processes are to be composed, in order to jointly reach their respective business goals. This is usually intended to be achieved by means of particular software constructs, denoted as *orchestrations* and *choreographies*. These notions need to be disambiguated. Furthermore, it is debatable whether it is a better choice to model business

processes together with interacting facilities and to do – on a basic level of argument – without orchestrations and choreographies.

Semantics of services: A lot of questions arise in analogy to programming languages, including the fundamental one: What is the semantics of a service, specified in a given modelling language? This is particular challenging, as some components of languages may not be intended to be implemented and thus deserve no formal semantics. Nevertheless, a notion of *equivalence* is required also in this case.

Expressive power of modelling languages: Again, this is a challenging problem, in particular due to the "always on" principle of services. In this context it is useful to identify the bare minimum of expressive power needed to specify services (and, consequently, the concepts common to all modelling languages).

Substitutability of services: This is a hairy problem, because long running processes (e.g. insurance contracts) must be substitutable during execution.

Brokering: The "SOA triangle" assumes a *broker* to match offerings of service providers with requirements of service requesters. This requires abstract information about processes and their capabilities from a user perspective: Which kind of information should the provider and the requestor disclose about their offered and requested services? This topic must be discussed on the level of models before it goes to implementation.

Relation between abstract and implementable processes: This topic has been partly addressed by extensions such as "BPEL for People", but deserves more investigation on the model level.

Reliability and Correctness: Properties of single and of composed services must be formulated and verified, to varying levels of rigour, on their models.

Monitoring and conformance: The realization and analysis of SOA is important, but only one side of the coin. The analysis of services while they are running is at least as important. It is vital to continuously monitor services to see whether they behave as expected. Moreover, the conformance of the real-life enactment and the normative models should be measured to detect mismatches and anomalies.

Discovery and mining: Process and data mining techniques can be used to extract information from interacting services. Process discovery, protocol mining, network analysis, etc. can be used to extract models from recorded behaviour.

Design Methodology: Methods and principles of Software Engineering must be adapted to SOA, on the level of models.

The above questions are both fundamental for SOA and far from being solved.

In the rest of this paper I suggest very first elements of an abstract basis of services. A generic modelling technique is exemplified in Chapter 3.

2. Some First Assumptions for a Formal Framework

Communication of services is conceptually to be organized as a service composition. Composed services together behave with regard to their joint environment like one service. For example, a travel agent composed with a flight carrier and a hotel may jointly offer week-end trips.

Each instance of communication between services may terminate. For example, a travel agent's communication with a client may terminate either with a contract, or with the understanding that the travel agent fails to meet the client's requirements. An example for irregular termination was a client's request never answered at all.

In a more refined setting, one may replace the requirement for termination by some more sophisticated "beauty predicate", or by additional conditions. Summing up, in the set S of all services we assume a binary composition operator

$$\oplus: S \times S \rightarrow S$$

Furthermore we assume a "beauty" predicate, i.e. a subset p of S.

In most cases, *p* denotes weak termination. This already lays the ground for a canonical, rich theory of services, covering a wealth of important notions, questions and properties, as they are particular important in the framework of SOA.

Two services R and S are *partners* iff their composition meets the "beauty" predicate, i.e. if

$$R \oplus S \text{ is an element of } p:$$

As services are made to communicate, i.e. are to be composed, the semantics of a service S is the set

$$sem(S) =_{def} \{R \mid R \text{ is a partner of } S\}$$

of all partners of S.

The fundamental notion of partners of a service S gives rise to a lot of questions:

- Composability*:* Given another service R, is R a partner of S?
- *Controllability:* Does S have partners at all?
- *Most liberal partner:* Is there a canonical partner of S?
- *Operation guideline:* How characterise all partners of S?
- *Adapter generation:* Given another service R, construct a service T such that R is a partner of $S \oplus T$.

The above definition of the semantics of a service implies a canonical comparison of the comprehensiveness of the capabilities of services: The capabilities of a services R are at most *as comprehensive as* those of S (written $R \leq S$) iff each partner of R is a partner of S. Hence, comprehension is a partial order on the set of all services, defined by

$$R \leq S \text{ iff } sem(R) \text{ is a subset of } sem(S).$$

A typical step towards a more comprehensive service was the above travel agent, additionally offering ship cruises.

Consequently, two services are *equivalent* iff they comprehend each other, i.e. iff they have the same partners. This equivalence is in fact the canonical counterpart of functional equivalence in the classical setting: Two systems are equivalent whenever their environment can not distinguish them.

3. A Modelling Technique for Services

The above considerations provided a conceptual framework which is now to be substantiated by a concrete modelling technique for services. It seems obvious from the previous considerations that programming languages are no adequate candidates for this endeavour.

Modelling techniques such as BPMN tend to a maximum of expressivity, for convenience to its users, resulting in a large number of concepts and graphical symbols. Here we follow the opposite direction, asking for the bare minimum of notions needed to model SOAs. The resulting *generic* technique highlights the characteristics of services and their composition. Furthermore, it is supported by nontrivial analysis techniques. The most widespread such techniques are based on Petri Nets. This is motivated by the observation that the communication style of services, i.e. asynchronous message passing, is perfectly met by the semantics of places of Petri Nets. Consequently, a service S can be modelled as a Petri Net with distinguished places to model the interface of S.

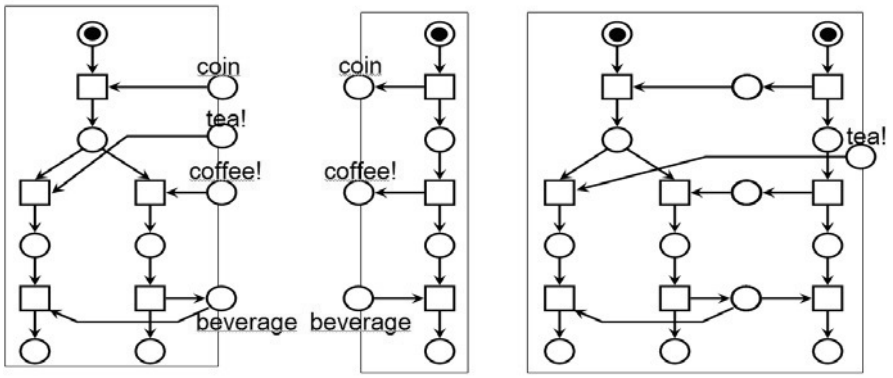


Fig. 1 Vending machine, coffee partner, and their composition

Figures 1 (a) and (b) show tow typical examples of services. Their intuitive meaning is obvious: (a) describes the service of a vending machine which expects its partners first to drop a coin and then to press a button, selecting coffee or tea. Finally, the vending machine provides the drink. Figure 1 (b) shows a corresponding partner. Finally, (c) shows the composition of the vending machine with its partner. The composed system terminates with tokens in terminal places.

Here we apply the usual conventions for graphical representations for Petri Nets with the interfaces places on the surface of an enclosing box. For the composition R⊕S of two nets R and S we assume w.l.o.g. that R and S are disjoint except for interface places. The sets of places, transitions, arcs and the initial marking of R⊕S are just the union of the corresponding sets of R and S. An out-place of R which is coincidently an in-place of S (and vice versa, an in-place of R which is coincidently an out-place of S) turns into an inner place of R⊕S. Figure 1 (c) shows the composition of the two nets as given in (a) and (b).

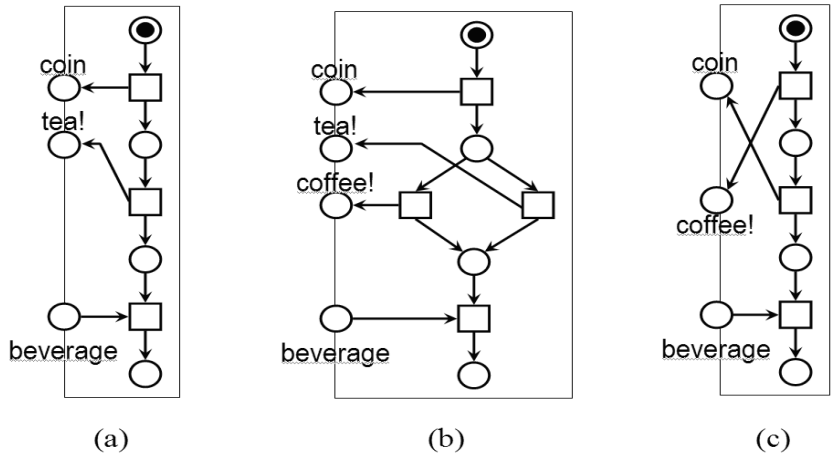


Fig. 2 Three more partners

Besides the partner shown in Fig. 1 (b) the vending machine has many more partners. Figure 2 (a) shows the tea-partner; (b) shows the coffee-or-tea-partner, and (c) shows that a partner may swap the order of dropping a coin and selecting a beverage.

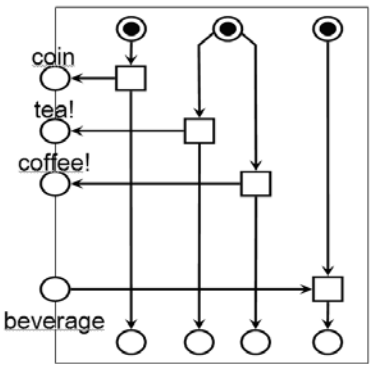


Fig. 3 Partner with three independent threads of control

There is no need for a service model to do with only one thread of control: Figure 3 shows a partner C with *three* threads of control. In fact, C is the *most permissive* partner of Fig. 1 (a): To derive any other partner form this one, extend the order in which actions occur in C, and fix one of the alternatives.

The *operating guideline* OG(S) of a service S describes all possible ways to make use of S. In the above technical framework this means OG(S) describes *all* partners, i.e. the semantics sem(S). In fact, OG(S) can finitely be described for each service S. Essentially, this description is an inscribed version of the most permissive partner of S.

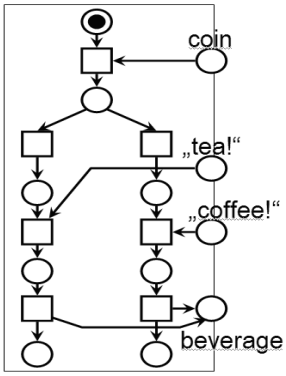


Fig. 4 A variant of the vending machine without partners

It is interesting to observe that there exist services with no partners at all: Figure 4 shows a variant of the vending machine which internally selects either tea or coffee. A partner would have to be able to correctly "guess"" this selection.

This kind of Petri Net models is expressive enough to define a feature complete semantics of the most important specification language for services, WS-BPEL [1],[2],[4],[6],[8],[9]. At the same time it is simple enough for a series of deep analysis techniques. For example, there are algorithmic solutions to all problems mentioned at the end of Chapter 3 [5].

4. Services with Ports

A theory of services must allow for abstraction. To this end we suggest ports as sets of interface places. Technically, the ports of a net S define a partition of the interface places of S, i.e. each interface place belongs to exactly one port. Furthermore, each port is named, with different ports of a service S named differently. For reasons to become reasonable later, each port is either an in-port or an out-port.

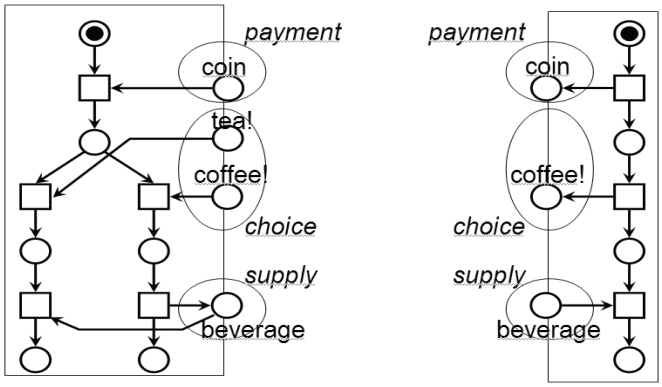


Fig. 5 Vending machine and coffee partner with ports

Figure 5 extends two nets of our running example by ports. Payment and choice are names of in-ports of the vending machine as well as out-ports of the coffee partner. Vice versa, supply is the name of an out-port of the vending machine, as well as of an in-port of the coffee partner. The composition R⊕S of two nets R and S with ports is again a net with ports.

As an example, the composition of the "plain" vending machine and coffee partners in Fig. 3 (b) and (c) resulted in the net (c) with "tea" an in-place. This is counter-intuitive. In fact, the composition of the versions with ports, as in Fig. 5, results in the intuitively satisfying open net of Fig. 6.

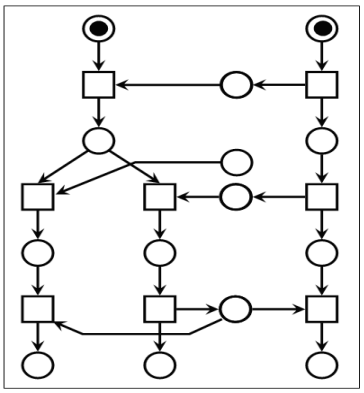


Fig. 6 Composition of the two nets of Fig. 7

5. Many Partners

A service (and hence a net) may serve more than one partner. As an example, the vending machine's partner in Fig. 3 can be dissolved into three partners, as in Fig. 7.

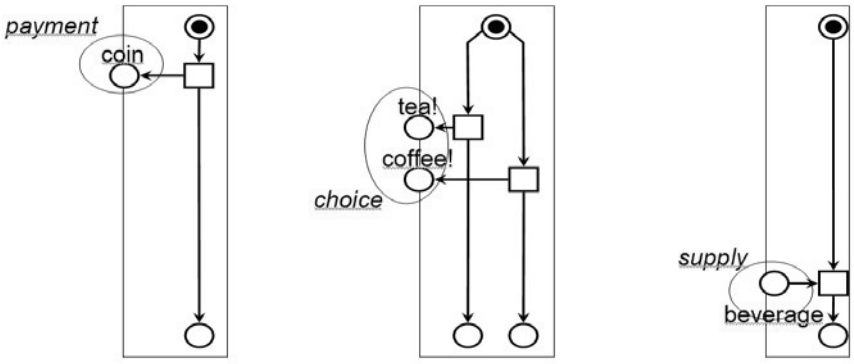


Fig. 7 Three partners of the vending machine

They together serve the port equipped version of the vending machine in Fig. 5. Each of the three nets can be constructed without considering the other two. This is not possible for all open nets with ports. As an example, Fig. 8 (a).

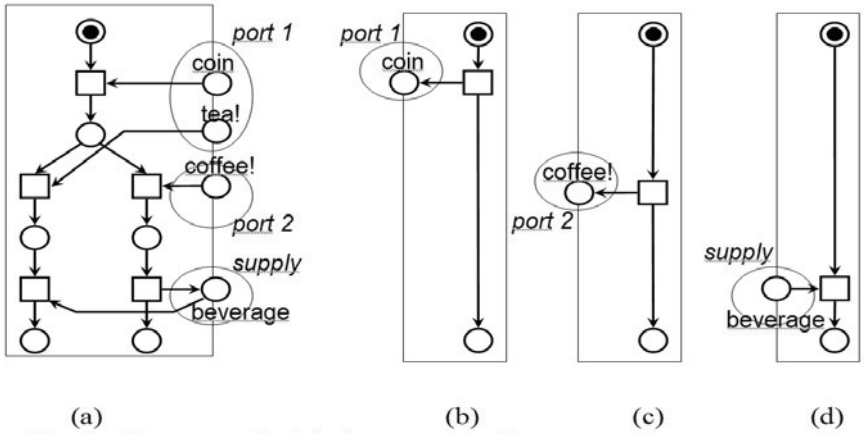


Fig. 8 Partners to be jointly constructed

References

1. Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. *Analyzing Interacting WS-BPEL Processes Using Flexible Model Generation.* Data Knowl. Eng., 64(1):38-54, January 2008.
2. Niels Lohmann and Jens Kleine. *Fully-automatic Translation of Open Workflow Net Models into Human-readable Abstract BPEL Processes.* Modellierung 2008, Proceedings, Lecture Notes in Informatics (LNI), 2008
3. Wolfgang Reisig, Karsten Wolf, Jan Bretschneider, Kathrin Kaschner, Niels Lohmann, Peter Massuthe, and Christian Stahl. *Challenges in a Service-Oriented World.* ERCIM News, 70:28-2, 2007.
4. Niels Lohmann. A *Feature-Complete Petri Net Semantics for WS-BPEL 2.0.* WS-FM LNCS, 2007, 5. Niels Lohmann, Peter Massuthe, and Karsten Wolf. *Behavioural Constraints for Services.* BPM 2007, LNCS 4714 pages 271-287,
5. Simon Moser, Axel Martens, Katharina Görlach, Wolfram Amme, and Artur Godlinski. *Advanced Verification of Distributed WS-BPEL Business Processes Incorporating CSSA-based Data Flow Analysis*. SCC 2007 pages 98-105, 2007.
6. Wolfgang Reisig, Jan Bretschneider, Dirk Fahland, Niels Lohmann, Peter Massuthe, and Christian Stahl. *Services as a Paradigm of Computation.* LNCS 4700 pages 521-538, 2007
7. Niels Lohmann, Peter Massuthe, Christian Stahl, and Daniela Weinberg. *Analyzing Interacting BPEL Processes.* BPM 2006, LNCS 2006.
8. Sebastian Hinz, Karsten Schmidt, and Christian Stahl. *Transforming BPEL to Petri Nets.* BPM 2005, LNCS 3649
9. Niels Lohmann, Oliver Kopp, Frank Leymann, and Wolfgang Reisig. Analyzing BPEL4Chor: Verification and Participant Synthesis. LNCS 4937

A contract-oriented view on threat modelling

Olav Skjelkvåle Ligaarden and Ketil Stølen
{olav.ligaarden,ketil.stolen}@sintef.no
SINTEF ICT and University of Oslo

Abstract

We propose a modular approach to the modelling and analysis of threat scenarios with mutual dependencies. Our approach may be used to deduce threat scenarios for a composite service from threat scenarios of its constituent services. It may also be used to decompose a threat scenario of a complex service into separate threat scenarios of its constituent services. A custom made contract-oriented paradigm to describe threat scenarios with external dependencies is put forward. We also provide a set of deduction rules facilitating various kinds of reasoning, including the analysis of mutual dependencies between threat scenarios.

1 Introduction

We present the approach in example-driven manner. For full documentation we refer to [1]. The example, "Survival of the SMSest", was orginally developed by Øystein Haugen as a compulsory exercise in the course INF5150 - Unassailable IT-systems at the University of Oslo in 2007.

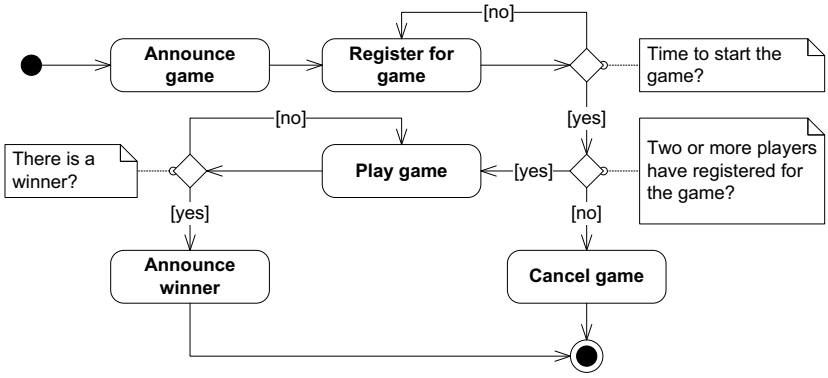


Figure 1: Activity diagram for the game *Survival of the SMSest*

In Figure 1 is an activity diagram for the game *Survival of the SMSest*. The game presented here is a modified version of the original game description given in [2]. This is a survival game which takes place within a given geographic area. The players play the game by using their mobile phones. In its simplicity, the players send SMS to strike out other players and to guard themselves against strikes from others. The diagram shows the different services that the game consists of, including the composite service "Play game", which will be the scope of our threat modelling. Before we present the constituent services of this composite service, we give a short description of the other services.

The service "Announce game" is used by the game administrator to announce a game to a set of potential players. The announcement contains information about the price (in EURO) for participating, the start time of the game and the geographic area where the game will take place. After the game has been announced, the players can use the service "Register for game" to register for the game. The players will pay the price for participating for which they will get an inital sum of points and an inital basic shield value. The game will start at the announced time if two or more players have registered. If not, the game will be cancelled and the registered players will get their money back. The service "Play game" will run until only one of the players is still alive. This player will be announced as the winner, and he will then cash-in his points in real money (1 EURO per point).

The composite service "Play game" consists of the following constituent services:

Light up area A player may light up an area around himself. The player receives a message naming all players within this area and their remaining points. A message is also sent to the players within this area that says that the first player has seen them. The cost (in points) of using this service depends on the size of the area that the player lights up.

Put up an extra protective shield Once spotted the player may choose to protect himself by putting up a shield. The cost (in points) of using this service depends on the intial strength, which will be gradually reduced over time, and the duration of the shield.

Strike A player may strike against another given player. The effect of the strike will depend upon its initial force, its duration and the distance between the striker and the stricken. A strike is fatal if the stricken player's shield is reduced to zero or below. The striker will then get the stricken player's points and boost his basic shield by some value, where the basic shield boost is based on the received points. The cost (in points) of using this service depends on the intial force, which will be gradually reduced over time, and the duration of the strike.

2 Threat models

Figure 2 shows an asset diagram on behalf of the gaming company on whose behalf the threat modelling is conducted. The three assets within the rectangular container are so-called direct assets. Each of these assets are related to the profit of one of the services, and harm to each of them can indirectly cause harm to the asset *Company profit*.

We aim to come up with a threat model for a composite service. Even in a relatively small case, such as this, there is a lot of information to process. We can simplify this processing by decomposing the threat modelling of the composite service into threat modelling for its constituent services. This can be done as long as we can combine these models in the end to get a valid model for the composite service as a whole. As we will see, one complicating factor here is that we can have threat models that are mutually dependent, meaning that one threat model depends on the others and vice versa. To compose the models we need means to distinguish mutual dependencies that are well-founded, meaning not circular, from mutual dependencies that are not.

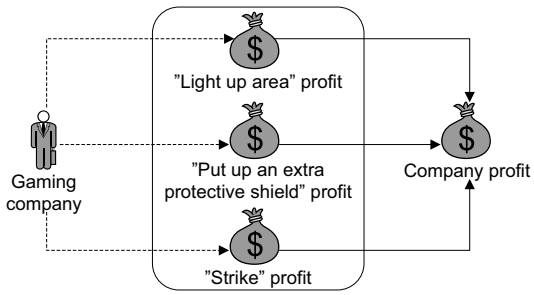


Figure 2: Asset diagram for the system

Figure 3 shows a threat diagram. This diagram takes into consideration the external threat scenario *The attacked player's score is too low and incorrect*, which is defined in the diagram in Figure 5. In the diagram in Figure 3 we refer to the content of the rectangular container including the crossing relation as the target scenario, and to the rest as the context scenario. We keep our assumptions about the context as generic as possible in order to facilitate reuse. We therefore leave open the likelihood of the assumed external event. Due to this we also need to leave open the likelihood of the event inside the target that is affected by the external event. In the textual syntax a threat diagram is written $C \rhd T$, where C and T are referred to as the context (which may be empty) and the target scenario, respectively. The semantical meaning of $C \rhd T$ is:

$$[\![C \rhd T]\!] := [\![T]\!] \text{ assuming } [\![C]\!] \text{ to the extent there are explicit dependencies}$$

The suffix 'to the extent there are explicit dependencies' is significant. It implies that if there are no relations connecting C to T explicitely, then we do not gain anything from C.

The threat diagram in Figure 3 focuses on the service "Light up area". This diagram describes how the threats *Eavesdropper* and *Hacker* can exploit vulnerabilities of the system in order to initiate the threat scenario *Sell critical player info to other players*. The *Hacker* can obtain critical player information from the game database, while the *Eavesdropper* can intercept SMS messages between players and the system to get access to critical player information. By critical information we mean information about a player's position, remaining points and shield value. The threat scenario may lead to other threat scenarios, which again may lead to unwanted incidents, which again may harm the direct asset *"Light up area" profit*.

The initiate relations (between a threat and a threat scenario or an unwanted incident), the threat scenarios, the leads to relations (between two threat scenarios, or two unwanted incident or one of both) and the unwanted incidents may be given a likelihood. The likelihood estimates are given by representatives of the customer, this case the gaming company, or based on historical data. It is also common to annotate harm relations (between an unwanted incident and an asset) with a consequence value, but that has not been done in our diagrams.

When a player knows the position and remaining points of other players he can move away from stronger players in his area or he can attack other players without lighting

up the area first. In both cases he will cheat by not using points to light up the area, which again will cause harm to the asset *"Light up area" profit*. The player may also move closer to other players before he lights up the area. In this way he can light up the area more effectively, and he will not draw the attention of other players since he lights up the area before he potentially attack, which is the normal protocol in the game. This unwanted incident will also cause harm to the asset. In the diagram we also have an external threat scenario which may lead to an unwanted incident within the target. In this scenario a player (the attacker) has received too few and an incorrect number of points after an attack, due to that the *Hacker* has changed critical information about the attacked player in the database (see the diagram in Figure 5). This may lead to an unwanted incident where the attacker has too few points and can therefore not light up the area, which again will harm the asset.

In Figure 4 we have a threat diagram for the service "Put up an extra protective shield". In this diagram we have a threat scenario where a player can make an assessment on how he should protect himself when a player lights up the area, based on that he knows the position and remaining points of that player. This will lead to an unwanted incident since the player can protect himself more effectively by cheating. In this diagram we also have two external threat scenarios from the diagram in Figure 3. The first threat scenario is that the player moves away from stronger players. This may lead to that another player lights up the area this player was previously in, which again will lead to an unwanted incident since the cheating player does not need to protect himself since he is not in this area anymore. In the second threat scenario a player may attack without lighting up the area first. This will lead to that the attacked player is not warned, which again will lead to an unwanted incident since the player cannot put up a defence, which again will affect the profit of this service. The third external threat scenario is the same as in the diagram in Figure 3. In this case the unwanted incident is that the attacker does not have enough points to defend himself.

The last threat diagram, which is for the service "Strike", is given in Figure 5. In this diagram the threat *Hacker* can not only obtain critical player information, be he can also exploit vulnerabilities in order to intiate the threat scenario *Change critical player information*. This may again lead to that an attacked player's score is too low and incorrect, which again may lead to that the attacker has not enough points to initiate a new strike, and this will again affect the profit of this service. Another threat scenario in this diagram is that a player knows the position of the player with the lowest shield and highest point score. An attack against this player will lead to an unwanted incident since the attacker is cheating and by doing so he uses an almost correct amount of points to attack and he earns a lot of points. This unwanted incident will affect the asset and it may also lead to another unwanted incident since the cheating player might win the game, which again will affect the profit of this service since the cheating player can cash-in his points in real money. The diagram also have two external threat scenarios, where both may lead to the same unwanted incident which is that the strike is more effective due to that the cheating player moved closer to the target or that the attacked player was not warned about the attack since the attacking player did not light up the area first.

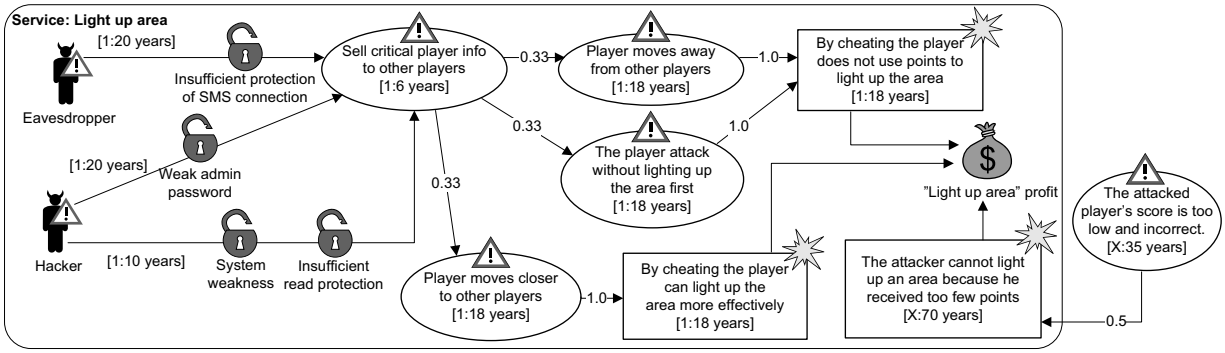


Figure 3: Threat diagram for the service "Light up area"

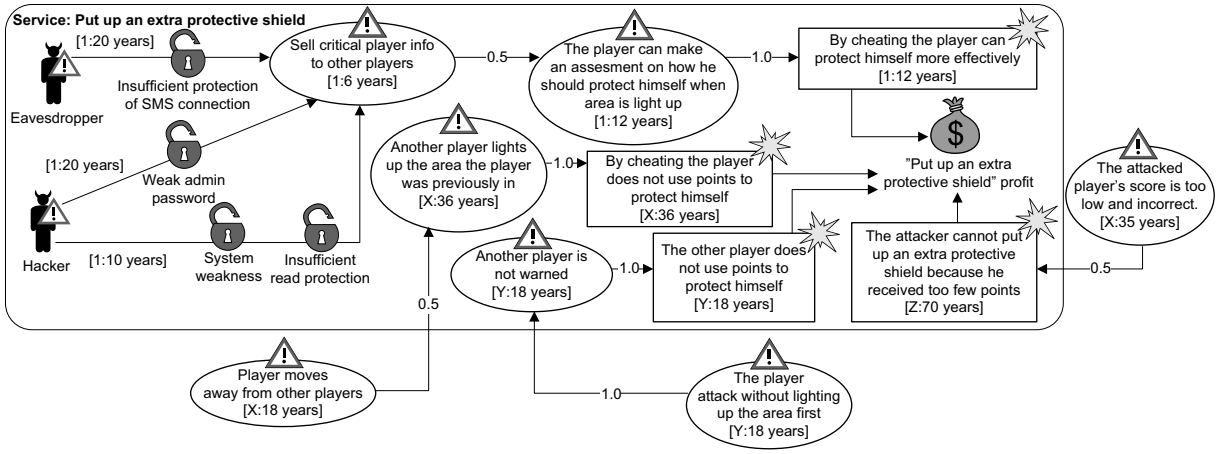


Figure 4: Threat diagram for the service "Put up an extra protective shield"

3 Reasoning about dependencies

By using Rules 5–8 (see the Appendix) in the calculus of [1] we may deduce the diagram for the composite service "Play game", shown in Figure 6, from the validity of the diagrams given in Figures 3–5. To be able to combine the three target scenarios it is vital that the paths of dependencies between the three diagrams are well-founded. By well-founded we mean that when we follow a path backwards we will cross the three diagrams only a finite number of times.

Or more rigorously, assume the validity of

$$C_1 \rhd T_1, \qquad C_2 \rhd T_2, \qquad C_3 \rhd T_3$$

obtained from the diagrams in Figures 3, 4 and 5, respectively, via the substitutions

$$\{X \mapsto 1\}, \qquad \{X \mapsto 1, Y \mapsto 1, Z \mapsto 1\}, \qquad \{X \mapsto 1, Y \mapsto 1\}$$

We want to deduce

$$\rhd T_1 \cup T_2 \cup T_3$$

which corresponds to the diagram in Figure 6. Here, we may understand the union operator on scenarios as a logical conjunction. Hence, from $\rhd S_1$, $\rhd S_2$ and $\rhd S_3$ we may deduce $\rhd S_1 \cup S_2 \cup S_3$, and the other way around.

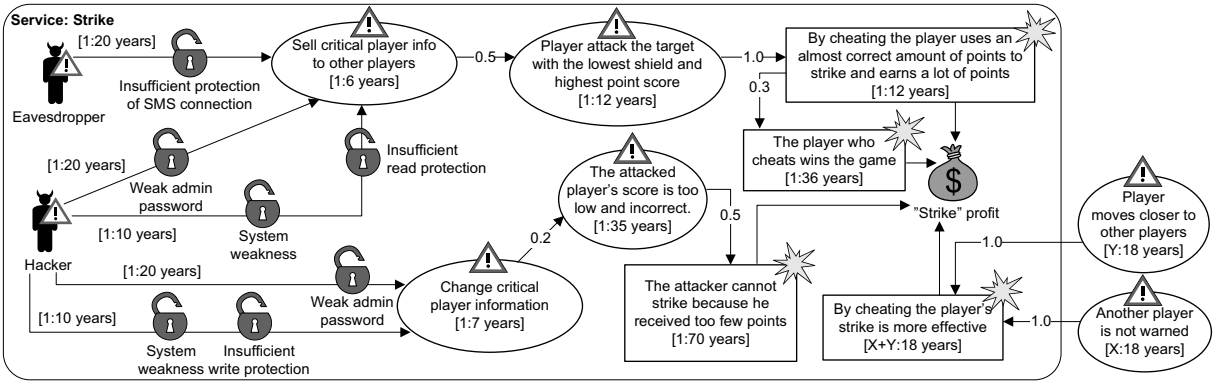


Figure 5: Threat diagram for the service "Strike"

The context C_2 can be decomposed into C_{21}, C_{22} and C_{23}, where C_{21} is connected to *Another player lights up the area the player was previously in*, C_{22} is connected to *Another player is not warned* and C_{23} is connected to *The attacker cannot put up an extra protective shield because he received too few points.* The context C_3 can be decomposed into C_{31} and C_{32}, where both C_{31} and C_{32} are connected to *By cheating the player's strike is more effective.*

We may use Rule 7 and 5 to deduce $\triangleright C_1$ from $C_3 \triangleright T_3$ since $C_1 \subset T_3$ and since C_1 does not depend on C_3. By doing so we have $\triangleright T_1$ by Rule 8. From $\triangleright T_1$ we can deduce $\triangleright C_{21}$, $\triangleright C_{22}$ and $\triangleright C_{32}$. We may deduce $\triangleright C_{32}$ from $\triangleright C_{22}$, which means that we have deduced $\triangleright C_3$. We have then $\triangleright T_3$ by Rule 8, and from $\triangleright T_3$ we can deduce $\triangleright C_{23}$. This means that we have deduced $\triangleright C_2$, and we have then $\triangleright T_2$ by Rule 8 in which case we have deduced the validity of the diagram in Figure 6.

4 Conclusion

We have presented the modular threat modelling approach from [1] in an example-driven manner. We have showed that we can decompose the threat modelling of a composite service into threat modelling of its constituent services, by making assumptions about the context explicit. We have also showed that these threat models can be combined in the end into a composite threat diagram for the composite service.

The research for this paper has been funded by the DIGIT (180052/S10) project of the Research Council of Norway.

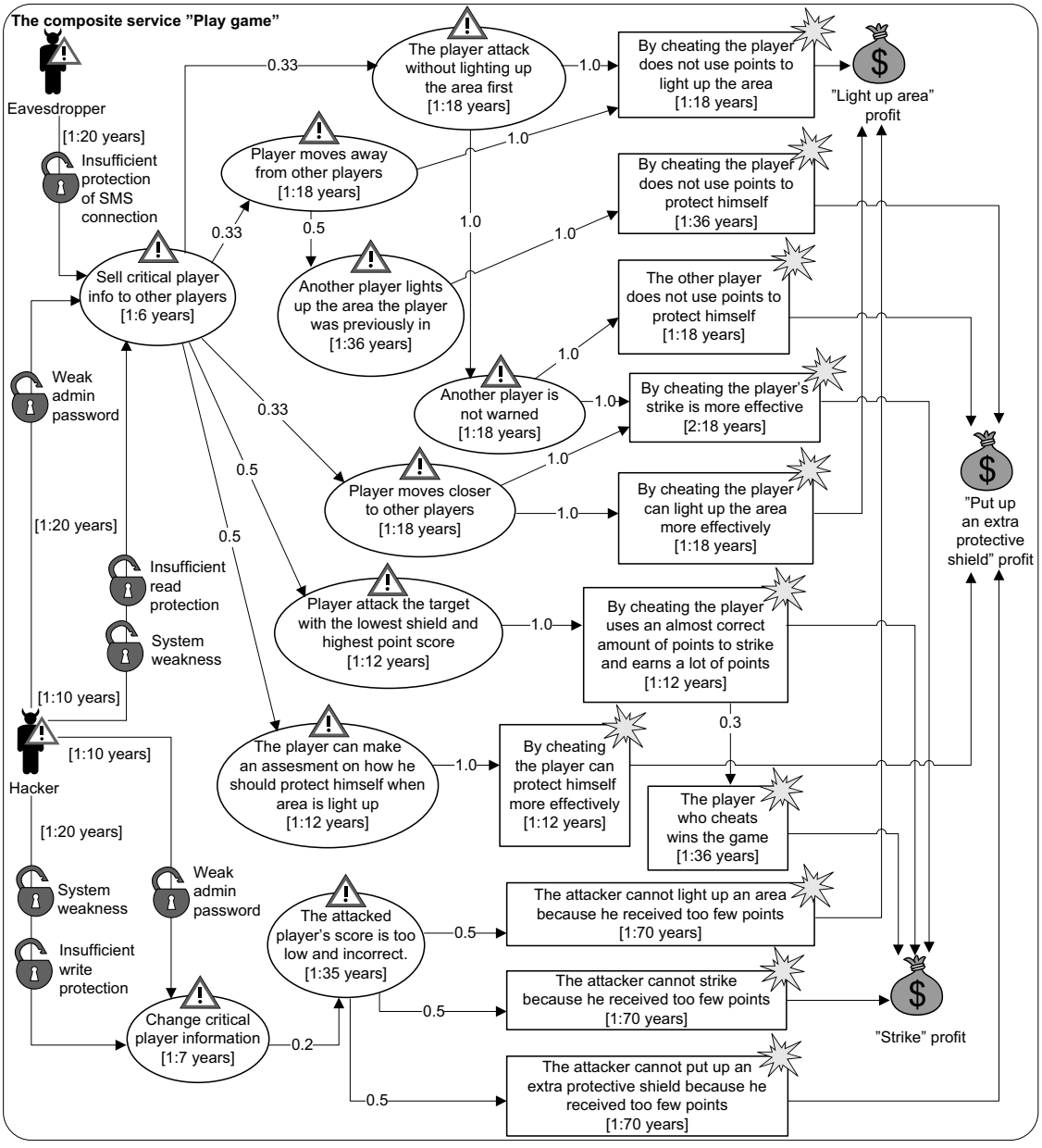


Figure 6: The composite service "Play game"

References

[1] Gyrd Brændeland, Heidi E. I. Dahl, and Ketil Stølen. Compositional risk analysis of mutually dependent systems. Technical Report A8360, SINTEF ICT, 2008.

[2] Øystein Haugen. "Survival of the SMSest" – the game, 2007. `http://www.uio.no/studier/emner/matnat/ifi/INF5150/h07/undervisningsmateriale/Oblig-2-INF5150-2007.pdf`.

Appendix

The Rules 5–8 in [1] are used to reason about dependencies. A short description of these rules are given in Table 1. More information about these rules can be found in [1].

Rule 5 (Independence):

$\frac{C \rhd T \quad C \ddagger T}{\rhd T}$ If we have deduced T assuming C, and T is independent of C, then we may deduce T. From the second premise it follows that there is no path from C to a vertex in T. Since the first premise states T assuming C we may deduce T.

Rule 6 (Context simplification):

$\frac{C \cup C' \rhd T \quad C \ddagger C' \cup T}{C' \rhd T}$ This rule allows us to remove a part of the context that is not connected to the rest. The second premise implies that there is no path from C to the rest of the diagram. Hence, the validity of the first premise does not depend upon C in which case the conclusion is also valid.

Rule 7 (Target simplification):

$\frac{C \rhd T \cup T' \quad T' \ddagger T}{C \rhd T}$ This rule allows us to remove part of the target scenario as long it is not situated in-between the context and the part of the target we want to keep. The second premise implies that there is no path from C to T via T'. Hence, the validity of the first premise implies the validity of the conclusion.

Rule 8 (Modus ponens):

$\frac{C \rhd T \quad \rhd C}{\rhd T}$ To make use of the rules 5–7, when scenarios are composed, we also need modus ponens for the $\rhd$-operator. Hence, if T holds assuming C to the extent there are explicit dependencies, and we can also show C, then it follows that T.

Table 1: Rules 5–8 in the calculus

Service Contracts in a Secure Middleware for Embedded Peer-to-Peer Systems[1]

F. Benigni, A. Brogi, S. Corfini, T. Fuentes

Department of Computer Science
University of Pisa, Italy

1. Introduction

Peer-to-peer (P2P) systems are distributed computing systems where all network elements act both as service consumers and service providers. Most P2P communication mechanisms are not based on pre-existing infrastructures, but rather on dynamic ad-hoc networks among peers [1]. Embedded Peer-to-Peer (EP2P) systems [2] introduce new challenges in the development of software for distributed systems. An EP2P system is a P2P system where small, low-powered, low-cost embedded devices cooperate in exchanging and processing information using wireless channels. EP2P systems can be employed in a number of different application areas, including mobile telephony, home systems, or environmental monitoring. EP2P systems present a high degree of heterogeneity (applications may run on different devices, from PDAs to sensor network nodes, with quite different network bandwidth and computing power) and autonomy (the devices enter and exit the system in an independent way, calling for frequent reorganisations of the system).

One of the keys for the successful development of EP2P systems is the possibility of suitably abstracting from low level P2P issues (such as the continuously changing network topology, and the connections and disconnections of peers) by means of convenient middleware. The goal of the Secure Middleware for Embedded Peer-To-Peer Systems (SMEPP) European Project (www.smepp.org, [3]) is precisely to develop such a middleware, that will have to be secure, generic, and adaptable to different devices (from PDAs and smart phones to embedded sensor actuator systems) and to different domains (from critical systems to consumer entertainment).

One of the objectives of SMEPP is to feature a high-level, service-oriented model to program the interaction among peers, thus hiding low-level details that concern the supporting infrastructure. SMEPP services are associated with service contracts - which provide standard descriptions of SMEPP services - and with service groundings - which provide details on how to interoperate with services (i.e., communication protocols, message formats, port numbers, etc.). Service contracts are the key ingredients of the SMEPP mechanisms for service publication and discovery.

In this paper, after briefly introducing the main features of the SMEPP model, we describe the structure of service contracts that has been defined in the SMEPP project.

2. The SMEPP model

The design of the SMEPP service-oriented model has been driven by a thorough analysis of the middleware, security, and application requirements that were identified during the first year[2] of the project [4,5,6].

[1] Work partly supported by the SMEPP project (EU-FP6-IST0333563).
[2] The SMEPP project is a 3 year project that started in September 2006.

The key features of the model are the notion of *group of peers*, the notion of *service* offered by peers (or by groups), and the concern for *security*. In short, the model defines a set of security-aware primitives for peer management (e.g., to create peers), for group management (e.g., to create, join, or leave groups), for service management (e.g., to publish, unpublish, or discover services), and for message and event handling (viz., to send or receive messages, or subscribe, unsubscribe, raise, and receive events), to be implemented by one or more APIs[3]. Such primitives are the basic bricks for constructing the code of P2P entities[4].

2.1 SMEPP primitives

Because of space limitations, we only list here (Figure 1[5]) the set of SMEPP primitives available to software developers. A detailed description of the SMEPP primitives and of the SMEPP model can be found in [7,8]. We only outline here that service management primitives include a **`publish`** primitive (to publish a service contract in a SMEPP group), a **`getServices`** primitive (to identify the published services that match a given contract template), and a **`getServiceContract`** primitive (to retrieve an actual service contract).

```
// Peer Management Primitives
peerId newPeer(credentials)
peerId getPeerId(id?)
// Group Management Primitives
groupId createGroup(groupDescription,credentials)
groupId[] getGroups(groupDescription?,credentials)
groupDescription getGroupDescription(groupId,credentials)
void joinGroup(groupId,credentials)
void leaveGroup(groupId)
groupId[] getIncludingGroups()
groupId getPublishingGroup(id)
peerId[] getPeers(groupId?,credentials)
// Service Management Primitives
<groupServiceId,peerServiceId> publish(groupId,serviceContract,serviceGrounding)
void unpublish(peerServiceId)
<groupId?,groupServiceId,peerServiceId>[]
          getServices(groupId?,peerId?,serviceContract?,maxResults?,credentials)
serviceContract getServiceContract(serviceId)
sessionId startSession(serviceId)
// Message Management Primitives
output? invoke(id,operationName,input?,returnResult?)
<callerId,input?> receiveMessage(operationName)
void reply(callerId,operationName,output?,faultName?)
output receiveResponse(id,operationName)
// Event Management Primitives
void event(groupId?,eventName,input?)
<callerId,input?> receiveEvent(groupId?,eventName)
void subscribe(eventName?,groupId?)
void unsubscribe(eventName?,groupId?)
```

Figure 1. The SMEPP primitives.

2.2 SMEPP modelling language

The SMEPP model is equipped with a high-level language (SMoL – SMEPP Modelling Language) for specifying how to orchestrate SMEPP primitives into peer or service code. The availability of a

[3] The reference implementation (currently under development) is Java-based.
[4] We shall use the term "entity" to refer to peers or services.
[5] The question mark denotes optional parameters, square brackets represent arrays, and angle brackets composite data structures.

high-level specification language notably simplifies the time-consuming and error-prone task of specifying the interactions of a complex P2P system. Most importantly, the definition of formal semantics for such a language [8,9] enables the simulation and the analysis of the behaviour of peers and services, thus featuring the possibility of developing not only secure, but also a priori verified SMEPP specifications. Furthemore, the availability of automatic translators (e.g., the prototype SMoL2Java compiler) greatly simplifies the generation of executable code, which can be further completed to express data-related details of peer/service behaviour.

SMoL defines the behaviour of the SMEPP entities as compositions of basic commands into structured ones. SMoL is inspired by version 2.0 of BPEL [10], which recently became the OASIS standard for describing Web service compositions. A BPEL process describes the behaviour of a Web service that orchestrates one or more WSDL [11] services, and in turn, it exposes a WSDL interface to its clients. Similarly to BPEL, SMoL aims to describe both abstract and executable entity behaviour. The former is an abstract presentation of the service concrete behaviour (e.g., the abstract behaviour of a Java service), that can be exposed to potential clients. The latter serves to describe the actual executable behaviour, which can be executed in dedicated SMoL engines. Since the BPEL semantics [12] is quite complex (e.g., due to synchronisation links and dead-path elimination), the analysis of (interactions of) BPEL processes is both troublesome and very time consuming. Furthermore, the SMEPP requirements do not request several BPEL constructs (concepts). Therefore, SMoL has been designed from BPEL basically by removing the following BPEL concepts: compensations, synchronisation links (and hence dead-path-elimination), the *forEach* construct, serializable scopes, partner links, message properties, and correlation sets.

Because of space limitations, we only list here (Figures 2 and 3) the basic and structured commands of SMoL. A detailed description of SMoL can be found in [7,8].

```
primitive invocation
void empty()
void wait(for?, until?, repeatEvery?)
void throw(faultName, faultVariable?)
faultVariable? catch(faultName)
<faultName, faultVariable?> catchAll()
void exit()
```

Figure 2. SMoL basic commands.

```
COM ::= BasicCommand |
        Sequence COM+ EndSequence |
        Flow COM+ EndFlow |
        While boolCond COM EndWhile |
        RepeatUntil boolCond COM EndRepeatUntil |
        If boolCond COM Else COM EndIf |
        Assign [Copy FROM TO EndCopy]+ EndAssign |
        Pick [pickGuard COM]+ EndPick |
        InformationHandler COM [infoGuard COM]+ EndInformationHandler |
        FaultHandler COM [catchGuard COM]+ EndFaultHandler
boolCond ::= logicalExpression
FROM ::= variable | expression | literal | opaque
TO ::= variable
pickGuard ::= guard | wait(for?,until?)
infoGuard ::= guard | wait(for?,until?,repeatEvery?)
guard ::= receiveMessage(operationName) |
          receiveResponse(id,operationName) |
          receiveEvent(groupId?,eventName)
catchGuard ::= catch(faultName) | catchAll()
```

Figure 3. SMoL structured commands.

3. SMEPP service contracts

SMEPP services have contracts, groundings and implementations. The contract provides descriptive information on the service, while the implementation is the executable service (e.g., a C++ service) exposed to the middleware through a grounding.

A service contract describes "what the service does" (viz., the service signature), "how it does it" (viz., the service behaviour), and it may include other extra-functional service properties (e.g., QoS). The signature provides an abstract description of the operations offered by the service to its clients, and of the events raised by the service. The signature is necessary for the service invocation. The behaviour is described by means of a SMoL specification, that is, an orchestration of SMEPP primitives. The description of the behaviour is optional[6] and it serves, on the one hand, to match service contracts, and on the other hand, to analyse (e.g., to simulate) the functioning of entities, or their interactions with other entities.

The core of the SMEPP service-oriented model borrows concepts from state-of-the-art Web service technologies. On the one hand, we model service contracts using XML schemas and, in particular, we model service signatures similarly to WSDL [11] interfaces, and ontology information using the Ontology Web Language (OWL [13]). On the other hand, we model service behaviour similarly to BPEL [10] processes.

Generally speaking, we partition services into two classes: *state-less* and *state-full* services. On the one hand, state-less services do not keep track of their interactions with clients. Clients can invoke the operations of such services one or more times and in any order. For example, a temperature monitoring service can be implemented as a simple state-less service that only offers one operation that returns the environment temperature. Entities can then invoke this operation every time they wish to get a reading of the temperature. On the other hand, state-full services keep track of their interactions with clients. We divide state-full services into *session-less* and *session-full* services. Intuitively speaking, session-less (state-full) services are services that feature a single *virtual communication channel*, which is shared by all clients, and which is available since the service is published until the service is unpublished. Session-less services suitably model shared resources such as a shared whiteboard where every client can sketch at anytime. Session-full services instead maintain one channel, and one interaction state, per client. For instance, a remote calculator simultaneously serving several clients can be provided as a session-full service. The following Figure summarizes how the three types of services can be classified according to the concepts of interaction state and session management.

	Without interaction state	*With interaction state*
Managed without sessions	state-less	session-less
Managed with sessions		session-full

Figure 4. SMEPP service types.

As we will see next, while contracts always declare the type of behaviour of a service (state-less, session-less or session-full), the specification of the behaviour via a SMoL specification is optional in contracts.

[6] We use the term *behaviour-less* to refer to services that do not expose behaviour information in their contracts. Dually, we use the term *behaviour-full* to refer to services that expose behaviour information in their contracts.

3.1 Structure of SMEPP service contracts

According to the SMEPP requirements, SMEPP service contracts must be:

- Multilanguage and multiplatform: The same contract must be consumed by all the SMEPP implementations (e.g., Java, NesC) and also by all the considered devices (e.g., laptops, PDAs, smart phones).
- Simple and light: Contracts must be easily downloaded and processed by small devices in [E]P2P environments.
- Extensible and easy to manage: The contract definition must be easily extensible but at the same time it should remain compatible with the previous versions. The contract will be used to discover services, hence it should be easy to manage.

According to the aforementioned requirements, a SMEPP service contract is expressed using XML and its structure is validated by an XML schema file [7]. The structure of a SMEPP contract contains the following elements:

- **Profile.** The profile of the service, which is mandatory, defines basic information on the service, such as the service name and the service category.
- **Signature.** The signature of the service is mandatory and it includes:
    - Operations description. For each operation provided by the service:
        - operation name,
        - operation type (one-way, request-response),
        - input parameters description (name, type, possibly other extra information),
        - output parameters description (name, type, possibly other extra information),
        - list of exceptions (faults) possibly raised by the operation.
    - Type declarations.
    - Optional ontological annotations.
    - Possibly other additional information (e.g., other service documentation).

  Signature information is expressed with a reduced version of WSDL 2.0 [11]. SMEPP supports two types of operations: one-way (corresponding to the wsdl:in-only message exchange pattern) and request-response (corresponding to the wsdl:in-out message exchange pattern).
- **Behaviour**. The service behaviour specifies the interaction protocol that the service follows, in order for the client to correctly interact with it. The service behavior must at least specify the service type (state-less, session-less, session-full), and it can set an upper bound to the number of running sessions (in the case of a session-full service). The service behaviour optionally includes a (possibly partial) specification of the workflow representing the service execution, presented using SMoL. The SMoL service behavior specification is included in the contract by means of an XML schema plugged-in into the basic Contract.xsd.
- **Properties.** Optionally the service contract can include additional information helping to categorize the service according to different criteria (e.g., geographical, business type etc.). Supplementing the service contract with these values provides useful metadata and context that can be exploited to discover and consume the services. The categorization of a service is expressed with a list of properties, each specifying *category*, *name*, and *value* of the property. The category can be a reference to a taxonomy defined in a separate file.
- **QoS.** Optionally the contract can include information describing the Quality of Service *(*QoS*)* offered by the service. The *QoS* information must be expressed in a machine understandable format, so that the middleware can include QoS monitoring tools to verify how the service is fulfilling the expected QoS. Contract express QoS with a fixed XML schema that defines some QoS parameters and the relations among them, and it also allows the use of optional ontological annotations to improve the semantic information related to the QoS. For each QoSParameter the following information can be processed:
    - *Name*: The name of the QoSParameter
    - *Domain*: The domain or classification of the information contained in by the QoSParameter (e.g., Runtime-related, Transaction-Support, Security-Level, Cost-related, etc.).
    - *Nature*: The way in which the value is computed (viz., *Dynamically*, *Statically*).

- *QoSImpact*: Describes the way in which a variation or unfulfillment of the QoSParameter would affect the performance and QoS of the service. In general it describes the influence (impact) that the QoSParameter produces over the whole service.
- *QoSMetric*: Describes the unit of measure and the way in which the QoSParameter can be measured. For example, the value '50' can be defined as a numeric type (e.g., xs:int), but the numeric value '50' may represent diverse concepts (percent, megabits per second, etc) and then QoSMetric can add some semantic meaning to '50' by means of the definition of its associated metric.
- *RelationShip*: Describes how this QoSParameter can affect other QoSParameters.
- *Aggregations*: Describes some compositional rules applied to the QoSParameter. Used to describe compound QoSparameter.
- Optional ontological annotations.

3.2 A simple example

The following is a very simple example illustrating the (friendly) syntax of the contract for a *TemperatureReader* service featuring the operation **`getTemp()`** which returns the current termperature.

:

```
Begin_Contract:

Profile:
       ServiceName: TemperatureReader
       ServiceCategory: EnvironmentMonitoring

Signature:
       Request-response operation temperature getTemp()

Behaviour:
       ServiceType: state-less
       Process:
            Sequence
                 <callerId> = receiveMessage("getTemp")
                 t = opaque // measure ambient temperature
                 reply(callerId, "getTemp", t)
            End Sequence

Properties:
       [Geography::Location] = "Italy";
       [Business::Functionality] = "Environmental Sensor"

QoS:
   -   [Transaction_Support::integrity] = 100'%'
   -   [Runtime_Related::latency] = 5'sec'
       latency produces an inverselyProportional impact over performance
   -   [Runtime_Related::throughput]  = 1000'request per hour'
       throughput produces a proportional impact over performance
   -   [Runtime_Related::performance]  = 100'%'
       performace is a compound QoSParameter composed by latency and throughput

End_Contract
```

Figure 5. Friendly representation of a SMEPP service contract.

Note that – for exemplification purposes – the above contract includes a SMoL specification of the service behaviour, even if the considered service is state-less. Note also that, for readability, we employed a friendly syntax for properties. For instance **`[Geografy::Location]="Italy"`** should be read as "the property named **`Location`** corresponding to the category **`Geography`** has the string value **`Italy`**". Similarly, **`[Runtime_Related::latency]=5'sec'`** should be read as "the

QoSParameter named **`latency`** corresponding to the *QoSDomain* **`Runtime_Related`** has the numeric value **`5`** and is measured in seconds (**'sec'**)."

4. Service grounding

The *service grounding* must include some metadata needed to correctly interoperate with the service. To let the SMEPP middleware use services which can be implemented in different ways using different platforms and architectures, we need to define a structure of grounding satisfying all the possible services architectures and implementations (e.g., component model, web services, remote objects, RPC, etc.). This section describes the service grounding specification for third party services, whilst SMEPP services will be accessed by the middleware by means of a standard interface which is part of the SMEPP API implementation.

The SMEPP middleware controls the instantiation and invocation of services into the [E]P2P environment, while SMEPP clients can interact with SMEPP services only through primitives (e.g., *getServices*, *invoke*, etc.) which do not allow the reception of any information about grounding (e.g., address, port number, etc). SMEPP grounding information is only managed in the provider's middleware, thus it cannot be used by clients to directly connect to SMEPP services. The XML schema of service groundings for third party services, sketched in Figure 6, is defined in the file Grounding.xsd.

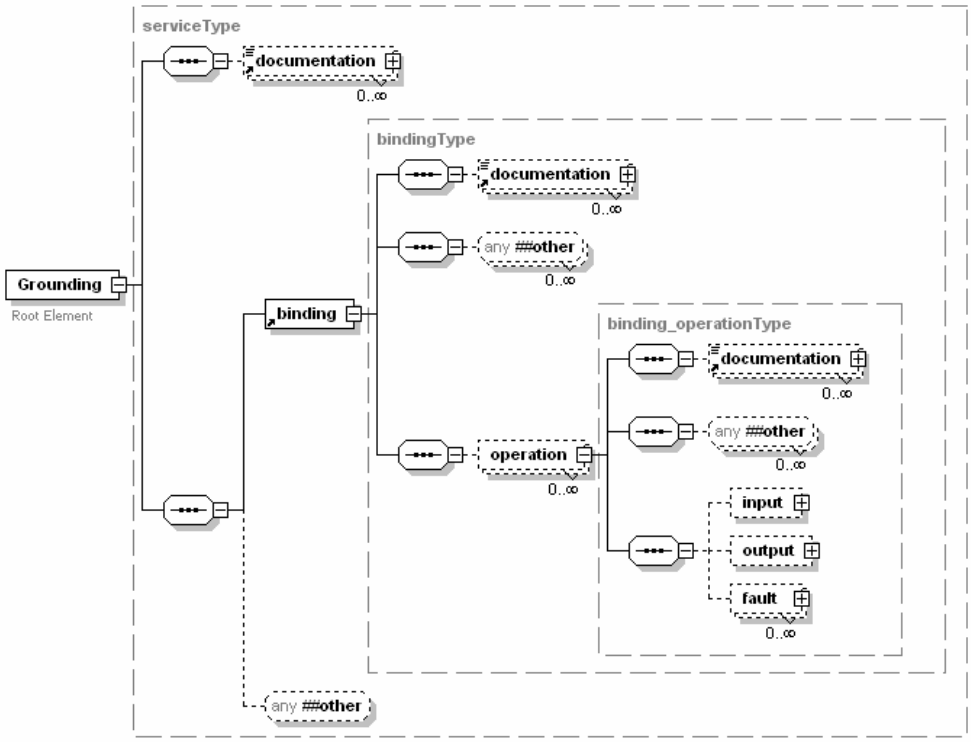


Figure 6. Representation of the Grounding.xsd schema.

The *binding* element of a SMEPP grounding is equivalent to the binding element of a WSDL document.

5. Concluding remarks

The way in which SMEPP contracts have been defined obviously bring similarities with other definitions of contracts that have been proposed in other research projects. For instance, signature information in SMEPP contracts is expressed with (a simplification of) WSDL 2.0 [11], while service behaviour is specified with SMoL, which is a simplification of BPEL [10]. While OWL-S [14] process models can be used to describe service behaviour, SMEPP does not employ OWL-S to represent service behaviour since OWL-S does not allow one to naturally model exception and event handling (which are instead a central part of SMoL and SMEPP, especially important for verification and analysis purposes). SMEPP instead shares with OWL-S the use of OWL ontologies to annotate concepts in contracts. SMEPP definition of QoS is instead borrowed from the Amigo project ("*Ambient intelligence for the networked home environment*") [15].

The SMEPP service discovery mechanism relies on service contracts. Queries (issued via the `getServices` primitive) employ partial contract specifications – also named *contract templates* –to restrict the set of candidate contracts to be retrieved. The current prototype implementation of the service discovery component of the SMEPP middleware supports syntactic queries – taking into account ontological annotations, if any - that can involve all parts of contracts but SMoL behavioural descriptions.

Immediate future activities are going to be devoted to experiment the resource requirements for the different types of devices participating in SMEPP applications, with the purpose of devising different types of matching for the different middleware configurations. Future work will also have to be devoted to develop tools for analysing the compatibility of behavioural descriptions, and to identify a suitable, less expressive language (e.g, behavioural types or even FSMs) to represent service protocols, in order to make their inclusion in the matching feasible in the context of SMEPP.

References

[1] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content addressable network," in Proceedings of SIG-COMM'01, 2001, pp. 161–172.

[2] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The runes middleware: A reconfigurable component-based approach to networked embedded systems. In Proceedings of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC'05), Berlin (Germany), Sept. 2005.

[3] M. Albano, A. Brogi, R. Popescu, M. Diaz, and J. Dianes. Towards secure middleware for embedded peer-to-peer systems: Objectives and requirements. In Proceedings of RSPSI'07, 2007, http://www.igd.fhg.de/igd-a1/RSPSI2/papers/Ubicomp2007_RSPSI2_Albano.pdf.

[4] SMEPP Coalition. D1.1: State of the art and generic middleware requirements. http://www.smepp.org/.

[5] SMEPP Coalition. D1.2: Security Requirements of EP2P Applications. http://www.smepp.org/.

[6] SMEPP Coalition. D1.3: Application requirements. http://www.smepp.org/.

[7] SMEPP Coalition. D2.1: Service model description. (Second version.) http://www.smepp.org/.

[8] A. Brogi, R. Popescu. Workflow Semantics of Peer and Service Behaviour. In J. Davies and X. Li (editors), Proceedings of the 2008 2nd IFIP/IEEE International Symposium on Theoretical Aspects of Software Engineering, pages 143-150, Nanjing, China, June 17-19, 2008.

[9] A. Brogi, R. Popescu, F. Gutierrez, P. Lopez, E. Pimentel. A service-oriented model for embedded peer-to-peer systems. Electronic Notes in Theoretical Computer Science, 194(4):5-22, 2008.

[10] OASIS.BPEL v2.0. http://www.oasis-open.org/committees/download.php/23974/wsbpel-v2.0-primer.pdf.

[11] W3C. Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. http://www.w3.org/TR/wsdl20.

[12] A. Brogi and R. Popescu. From BPEL processes to YAWL workflows. In M. Bravetti, M. Nunez, G. Zavattaro, editors, Proceedings of the 3rd International Workshop on Web Services and Formal Methods (WS-FM 06), LNCS vol. 4184, pages 107-122, 2006.

[13] D. McGuiness and F. van Harmelen (Eds). Web ontology language (OWL) overview, 2004. Web guide. http://www.w3.org/TR/owl-features.

[14] OWL-S Coalition. OWL-S 1.1 release, from http://www.daml.org/services/owl-s/1.1/.

[15] Amigo project, Deliverable 3.1b "Detailed Design of the Amigo Middleware Core: Service Specification, Interoperable Middleware Core". http://www.hitech-projects.com/euprojects/amigo/deliverable.htm

A Framework for Contract-Based Reasoning: Motivation and Application

Sophie Quinton and Susanne Graf

Université Joseph Fourier, VERIMAG

Abstract In [13], we have introduced a framework for contract-based reasoning parametrized, among others, by a notion of behaviors and a notion of composition of such behaviors. Our aim is to separate clearly between properties of contracts that are specific to a given framework and properties that apply to an entire class of contract-based verification frameworks. In particular, we think that soundness of (apparently) circular reasoning is a generic property that is essential when dealing with contracts. To support our claim, we discuss two particular instantiations of our framework that are adapted from [9] and [11]. We show that important results presented in these papers are direct consequences of the *circular reasoning* property of our framework. Proofs are thus shorter and, we believe, clearer. We also believe that such a methodology improves understanding of concepts. In particular, if circular reasoning is not sound in a given framework, there are two options: either not use it, or strengthen the definition of refinement in a context so that in the modified framework circular reasoning is sound.

1 Introduction and related work

Compositional verification is necessary to make verification scalable to systems of arbitrary complexity. One well-studied approach is Assume/Guarantee reasoning ([6]) which proposes proof methods that allow to verify a system – defined as a composition of components – by (1) expressing the global property as a composition of local properties and (2) verifying the local properties on individual components, taking into account some assumptions on their environment that must be proven to hold.

In the context of system design, completely refined models of the components may not be available at early stages of development, and one may want to reason about assumptions and properties only. We introduce for that purpose a notion of contract which distinguishes explicitly a constraint on the behavior of the environment – called assumption, a constraint on the behavior of the component under consideration – called guarantee (or promise) and a composition operator γ defining how the behavior of the component and its environment are intended to be composed. Notice that in most frameworks, γ is implicitly defined by the overall framework.

In [5], a notion of interface specification has been introduced in terms of I/O automata, where the constraint expressed on *inputs* is interpreted as the *assumption*, the constraint on outputs as the *guarantee* (under the given assumption), and γ is the intersection of prefix-closed traces. When dealing with such contracts, relevant notions are those of *composability* and *compatibility* of sets of contracts and of *refinement* between contracts. *Composability* is a purely syntactic criterion on interfaces and *compatibility* expresses the fact that there exists an environment preventing the composition to enter some predefined *error* state. Finally, in the context of contracts, *refinement* is sometimes also called *dominance* [2], and a refined contract is a contract that requires more from the implementation and less from the environment.

Separation between assumption and guarantee has been proposed in [9] while interface automata have been enriched with modalities in [11]. In [13] we have presented a framework generalizing these interface theories by adding a structural part to contracts as explained above (the composition operator γ). The composition operators and their composition are defined using the interaction layer of the BIP framework [7,1,3].

The framework BIP (Behavior, Interaction, Priority) has been proposed for component-based design and verification. In BIP, the concept of input/output-based interaction is replaced by the more general

concept of multi-party interaction which allows each of the involved interaction partners to impose constraints on when the interaction may take place and thus does not require input completeness. That means, contracts, as defined by interface specifications, are not needed to avoid *error* states. But it is still relevant to have a framework for assume/guarantee reasoning which allows to (1) characterize a set of environments E for which the composition $\gamma(K, E)$ is consistent, that is, deadlock-free, or (2) those for which $\gamma(K, E)$ guarantees that K behaves as specified by some property B where B may express a stronger property than a BIP behavior specification. This allows deriving global properties of a composed system in a compositional manner, as well as definition of frameworks for reusability and tailoring of components.

The motivation for the use of the BIP interaction models as such is their expressivity – as shown in [4], the BIP interaction model is as expressive as any set of (monotonic) SOS rules – and interesting properties of the composition of interaction models (such as associativity as shown in Figure 1). This will enable us to consider heterogeneous component systems, using a wide spectrum of composition operators, from asynchronous to fully synchronous composition, and which do not depend on a specific representation of behaviors.

In [13], as in [9], we clearly separate assumptions and guarantees, which are behaviors. This separation allows reuse of a component in a different setting and provides a better understanding of how responsibilities are divided between a component and its environment. As we make explicit the parameters of a reasoning framework, the need for different refinement relations playing different roles in the verification process appears: for example, in a contract-based verification approach, at least the following notions of refinement between two behaviors B_1 and B_2 have to be considered: B_1 refines B_2 in *a given context*, B_1 refines B_2 in *all contexts*, and B_1 refines B_2 if B_1 and B_2 are *closed systems*. Here, we do not introduce refinement between closed systems, as we are interested in incrementality, whereas refinement between closed systems is compositional for conjunction, but not incremental.

In this paper, we show the usefulness of the general framework for contract-based reasoning introduced in [13], by showing that both contract-based reasoning of [9] and [11] are specific instances of it. As we separate clearly between properties of contracts that are specific to a given framework and properties that apply to an entire class of contract-based verification frameworks (in particular circular reasoning), we are able to show that important results presented in these papers are direct consequences of the *circular reasoning* property of our framework. Proofs are thus shorter and, we believe, clearer. We also believe that such a methodology improves understanding of concepts. For example, if circular reasoning is not sound in a particular framework, one may either not use it, or strengthen the definition of refinement in a context so that in the modified framework circular reasoning is sound.

The paper is organized as follows: in section 2, we give an overview of the framework for contract-based reasoning of [13]. In section 3, we show how fitting interface I/O automata into our framework makes proofs shorter and clearer. In section 4 we explain how adding control information to interface modal automata as in [11] can be done naturally within our framework. We then conclude by explaining the benefits that we expect from using our framework as a design methodology.

2 A framework for contract-based verification

In this section, we first give an overview of the BIP framework. We then present the definition of *contract-based verification framework* of [13], and provide a sufficient condition for dominance between contracts.

2.1 The BIP interaction layer

BIP [7,1,3] is a component framework for constructing systems by separationg explicitly the *possible behavior* of individual components from an *interaction model* defining the set of allowed interactions and the data exchange realized by them, and a *priority preorder* defining arbitration between enabled interactions. This implies a clear separation between *behavior* and *composition structure*. As already

stated, BIP is general enough to encompass heterogeneity of interaction, allowing description of synchronous as well as asynchrous and heterogeneous systems.

We consider an abstract version of BIP without variable exchange on connectors nor priorities. The interface of a component is a set of ports, and a composition operator γ is defined by a set of *legal* interactions (multi-partner rendez-vous) given as a set of connectors of the algebra $\mathcal{AC}(P)$ introduced in [3]. We do not introduce the BIP interaction layer in detail, as it is not a prerequisit for understanding this paper. In figures, we represent ports that can *trigger* an interaction, that is, ports that are legal interactions by themselves, by a triangle-shaped connector end, while ports that need some interaction partner, called *synchrons*, are represented by a circle-shaped connector end. Notice that a connector consisting only of *synchrons* represents an n-ary rendez-vous.

The behavioral semantics of an operator γ depends on the representation of behaviors, but the composition of such γ is entirely defined within the BIP framework and it is assciative and commutative. This means that it is always possible to compute composition operators to compose a set of components in any order. For example, in Figure 1, the composition of B_1, E_1 and E_2 is represented in the left part of the figure as $\gamma_1(B_1, \gamma_2(B_2, B_3))$ and in the right part as $\gamma_3(\gamma_4(B_1, B_2), B_3))$. The expressivity of BIP interaction models has been discussed and compared to others in [4].

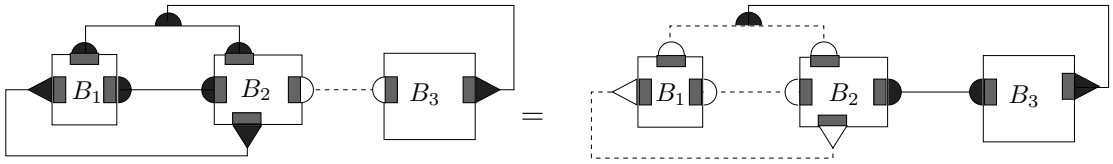


Figure1. BIP composition. In the left (resp. right) part, γ_1 (resp. γ_3) is represented by full lines with black connector ends and γ_2 (resp. γ_4) by dashed lines with white connector ends.

2.2 Defining a parametrized framework for verification

Definition 1 (Contract-based verification framework). *A contract-based verification framework is a tuple $(\mathcal{B}, \mathcal{P}, \Gamma, \|\,.\,\|, \theta)$, where:*

- *$\mathcal{B}$ is a set of behaviors; each behavior $B \in \mathcal{B}$ has as interface a set of ports denoted $\mathcal{P}_B$.*
- *$\mathcal{P} = \bigcup_{B \in \mathcal{B}} \mathcal{P}_{\mathcal{B}}$.*
- *Γ is a set of BIP composition operators defined on subsets of $\mathcal{P}$.*
- *$\|\,.\,\| \ : \ \Gamma \times 2^{\mathcal{B}} \longrightarrow \mathcal{B}$ is a partial function defining a behavior semantics for the composition of behaviors, ensuring associativity and commutativity of BIP composition operators as defined in [7]. For $\gamma \in \Gamma$ and $B_1, \ldots, B_n \in \mathcal{B}$, $\|(\gamma, (B_1, \ldots, B_n))\|$, denoted $\gamma(B_1, \ldots, B_n)$, is defined iff γ is defined on $\bigsqcup_{i=1}^{n} \mathcal{P}_{B_i}$.*
- *$\theta \ : \ \mathcal{B} \times \Gamma \longrightarrow 2^{\mathcal{B} \times \mathcal{B}}$ is a partial function that is* monotonic w.r.t. composition *as defined below. For each pair (E, γ) such that γ is a composition operator defined on $\mathcal{P}_E \sqcup \mathcal{P}_B$ for some set of ports $\mathcal{P}_B$, $\theta(E, \gamma)$, denoted $\sqsubseteq_{E,\gamma}$, is a preorder (a reflexive and transitive binary relation) over the set of behaviors with associated set of ports $\mathcal{P}_B$. θ is called* refinement in a context*.*

We say that $B \in \mathcal{B}$ is a behavior *on* $\mathcal{P}$ when $\mathcal{P}_B = \mathcal{P}$. In the following, we suppose a contract-based verification framework $(\mathcal{B}, \mathcal{P}, \Gamma, \|\,.\,\|, \theta)$ to be given.

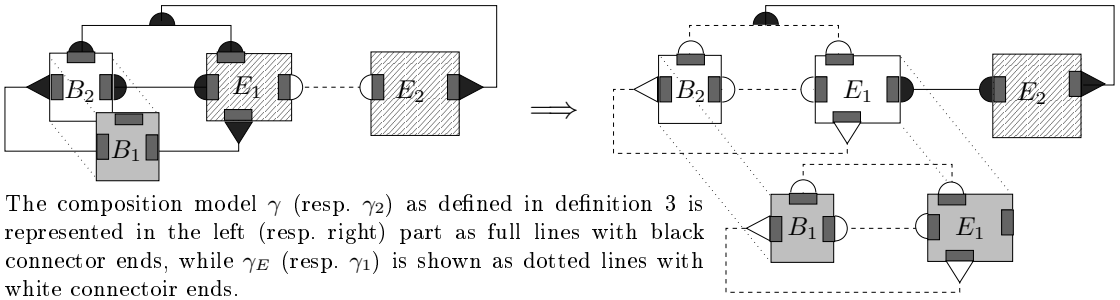
Definition 2 (Context for an interface). *Let $P \in 2^{\mathcal{P}}$ be an interface. A context for P is a pair (E, γ) where E is such that $P \cap \mathcal{P}_E = \emptyset$ and γ is a composition operator defined on $P \sqcup \mathcal{P}_E$.*

In this definition, γ constitutes the structural part of the context and E is its behavioral part.

Definition 3 (Monotony w.r.t. composition). *θ is monotonic w.r.t. composition iff the following implication always holds. Let $P \in 2^{\mathcal{P}}$ be an interface. Let (E, γ) be a context for P. If there exist γ_E on $\mathcal{P}_E$ and E_1, E_2 such that $\mathcal{P}_E = \mathcal{P}_{E_1} \sqcup \mathcal{P}_{E_2}$ and $E = \gamma_E(E_1, E_2)$, then for all B_1, B_2 behaviors on P:*

$$B_1 \sqsubseteq_{\gamma_E(E_1,E_2),\gamma} B_2 \implies \gamma_1(B_1, E_1) \sqsubseteq_{E_2,\gamma_2} \gamma_1(B_2, E_1)$$

where γ_1 and γ_2 are the composition operators calculated from γ and γ_E (see [I]) for respectively $P \sqcup \mathcal{P}_{E_1}$ and $P \sqcup \mathcal{P}_{E_1} \sqcup \mathcal{P}_{E_2}$.



The composition model γ (resp. γ_2) as defined in definition 3 is represented in the left (resp. right) part as full lines with black connector ends, while γ_E (resp. γ_1) is shown as dotted lines with white connectoir ends.

Figure2. Monotony of refinement.

Definition 4 (Refinement). *When for all E, γ, $B_1 \sqsubseteq_{E,\gamma} B_2$, we say that B_1 refines B_2 in all contexts, or simply B_1 refines B_2, and we use the notation $B_1 \sqsubseteq B_2$.*

Depending on whether B_1 and B_2 stand for implementations or properties, $B_1 \sqsubseteq_{E,\gamma} B_2$ can be interpreted in different ways: if B_1 is an implementation and B_2 is a property, it means that B_1 ensures B_2 in the context (E, γ); if both are properties then it means that B_1 implies B_2 in the context (E, γ), while if both are implementations $B_1 \sqsubseteq_{E,\gamma} B_2$ means that B_1 refines B_2 in the context (E, γ). We choose the term refinement.

Definition 5 (Contract for an interface). *Let $P \in 2^{\mathcal{P}}$ be an interface. A contract $\mathcal{C}$ for P consists of:*

- *a context $\mathcal{E} = (A, \gamma)$ for P, where A is called the* assumption
- *a behavior G on P called the* guarantee

We write $C = (A, \gamma, G)$ rather than $C = ((A, \gamma), G)$.

Assumptions and guarantees are defined on disjoint sets of ports. In some way, in a contract associated to a component (through its interface), the assumption is an abstraction of the environment and the guarantee an abstraction of the component. This allows easier reusability of components, as using them in a different environment requires only to modify the assumption. We say that a behavior B *satisfies* a contract $C = ((A, \gamma), G)$, denoted $B \models \mathcal{C}$, when $B \sqsubseteq_{A,\gamma} G$. Thus the satisfaction relation is a parameter of the verification framework as it corresponds directly to verification in a context.

Definition 6 (Dominance). *Let $\{P_i\}_{i=1}^n$ be a family of pairwise disjoint sets of ports, with $P = \bigsqcup_{i=1}^n P_i$. Let $\mathcal{C}$ be a contract for P, and for each $i = 1..n$, $\mathcal{C}_i$ be a contract for P_i. Let γ be a composition operator on P. We say that $\mathcal{C}$ dominates $\{\mathcal{C}_i\}_{i=1..n}$ w.r.t. γ iff $\forall B_1, \ldots, B_n \in \mathcal{B}$:*

$$(\forall i, P_{B_i} = P_i \ \wedge \ B_i \models \mathcal{C}_i) \implies \gamma(B_1, \ldots, B_n) \models \mathcal{C}$$

A contract $\mathcal{C}$ dominates a set of "subcontracts" $\{\mathcal{C}_i\}_{i=1..n}$ w.r.t a composition operator γ if any set of components satisfying the subcontracts, when composed using γ, makes a component satisfying $\mathcal{C}$. In particular, for $n = 1$, a contract $\mathcal{C}$ dominates a contract $\mathcal{C}'$ if every component satisfying $\mathcal{C}'$ also satisfies $\mathcal{C}$. Notice that the dominance relation is not defined between two contracts, but between a contract for an interface P and a set of contracts for a partition of P. Thus there is no need for a notion of least contract representing a composition of contracts.

These are all the notions required by any contract-based verification framework. We now focus on a specific property possibly attached to such a framework, namely soundness of circular reasoning, and its consequences.

2.3 A generic condition for dominance

Definition 7 ((Apparently) circular reasoning). *A framework $(\mathcal{B}, \mathcal{P}, \Gamma, \|\,.\,\|, \theta)$ allows apparent circular reasoning iff it is such that, given an interface $P \in 2^{\mathcal{P}}$, given a behavior B on P, a context (E, γ) for P and a contract $\mathcal{C} = (A, \gamma, G)$ for P, the following property holds:*

$$B \sqsubseteq_{A,\gamma} G \wedge E \sqsubseteq_{G,\gamma} A \implies B \sqsubseteq_{E,\gamma} G$$

This means that every component with a contract (A, γ, G), if integrated into an environment that is compatible with γ and that ensures A, can be replaced in the system by a component with the same interface and with G (or any refinement of it) as behavior. Everything that can be proved in the new system can also be proved in the original system.

We have shown in [13] an example of such a framework with behaviors expressed as modal transition systems ([8]). We also introduced the following proof rule that can be used to prove dominance between a contract and a set of subcontracts whenever the framework allows circular reasoning.

Theorem 1. *Let $(\mathcal{B}, \mathcal{P}, \Gamma, \|\,.\,\|, \theta)$ be a framework allowing circular reasoning. Let $\{P_i\}_{i=1}^n$ be a family of pairwise disjoint interfaces, with $P = \bigsqcup_{i=1}^n P_i$. Let $\mathcal{C} = (A, \gamma, G)$ be a contract for P, and for each $i = 1..n$, $\mathcal{C}_i = (A_i, \gamma_i, G_i)$ be a contract for P_i. Let γ_I be a composition operator on P. Then the following conditions are sufficient to prove that $\mathcal{C}$ dominates $\{\mathcal{C}_i\}_{i=1..n}$ w.r.t. γ:*

$$\begin{cases} \gamma_I(G_1, \ldots, G_n) \models \mathcal{C} \\ \forall i, \gamma(A, \gamma_{I\setminus i}(G_1, \ldots, G_{i-1}, G_{i+1}, \ldots, G_n)) \models \mathcal{C}^{-1} \end{cases}$$

with $\gamma_{I\setminus i}$ standing for the restriction of γ_I to $P\setminus P_i$ and $\mathcal{C}^{-1} = (G_i, \gamma_i, A_i)$.

3 Application to interface Input/Output automata

We explain here that the theory on interface Input/Output automata presented in [9] is an instantiation of the contract-based verification framework described in [13]. In particular, Theorems 3 and 4 of the original paper are direct consequences of the fact that the framework corresponding to interface I/O automata allows circular reasoning.

3.1 Presentation and terminology

An interface for a component in [9], which corresponds to our notion of contract, consists of two I/O automata, an assumption (called the *environment*) described on the ports of the environment and a guarantee (called a *specification*) on the ports of the component. Contrary to our approach, the interaction model between the component and its environment is predefined. As we can explicitely express the way component interact with each other, we do not need to distinguish between inputs and outputs in the behaviours, thus using usual LTSs instead of I/O automata. Our notion of refinement in a context is called *implementation of an interface* and denoted $E \models B \leq G$ for $B \sqsubseteq_{E,\gamma} G$.

3.2 Interface I/O automata as a contract-based verification framework

We define here a contract-based verification framework $(\mathcal{B}, \mathcal{P}, \Gamma, \|\,.\,\|, \sqsubseteq)$ that corresponds to interface I/O automata.

- $\mathcal{B}$ is a set of LTSs. For each LTS B, the set of its labels is denoted $\mathcal{P}_B$.
- $\mathcal{P} = \bigcup_{B \in \mathcal{B}} \mathcal{P}_{\mathcal{B}}$.
- Γ is the set of composition operators such that every connector has at most one trigger.
- $\|\,.\,\|$ is the standard BIP composition semantics for LTSs.
- For $E, B_1, B_2 \in \mathcal{B}$ such that $\mathcal{P}_{B_1} = \mathcal{P}_{B_2}$ and $\gamma \in \Gamma$ defined on $\mathcal{P}_E \sqcup \mathcal{P}_{B_1}$, $B_1 \sqsubseteq_{E,\gamma} B_2$ is defined as $Tr(\gamma(B_1, E)) \upharpoonright \gamma \subseteq Tr(\gamma(B_2, E)) \upharpoonright \gamma$, where $Tr(B)$ denotes the set of traces of B and $\upharpoonright \gamma$ is the projection of a set of traces onto γ.

The composition defined here is more general than the one proposed in [9], since we do not require that at most two components share an action. We do not apply hiding, as it is always possible to connect a new component to a connector. We only suppose that a connector has at most one trigger (output port). The projection $\upharpoonright \gamma$ onto the set of interactions between the component and its environment allows not to take internal actions into account when considering refinement.

In order to apply our methodology, we need to prove that $(\mathcal{B}, \mathcal{P}, \Gamma, \|\,.\,\|, \sqsubseteq)$ is indeed a contract-based verification framework. The binary relation $\sqsubseteq_{E,\gamma}$ is obviously transitive and reflexive for any E and γ. As we use the standard BIP composition of LTSs, we also know that associativity and commutativity are preserved.

Theorem 2 (Compatibility). *$\sqsubseteq$ as defined here is compatible with composition.*

Proof.
$$\begin{aligned} B_1 \sqsubseteq_{\gamma_E(E_1,E_2),\gamma} B_2 &\iff Tr(\gamma(B_1, \gamma_E(E_1, E_2))) \upharpoonright \gamma \subseteq Tr(\gamma(B_2, \gamma_E(E_1, E_2))) \upharpoonright \gamma \\ &\iff Tr(\gamma_2(\gamma_1(B_1, E_1), E_2)) \upharpoonright \gamma \subseteq Tr(\gamma_2(\gamma_1(B_2, E_1), E_2)) \upharpoonright \gamma \\ &\iff \gamma_1(B_1, E_1) \sqsubseteq_{E_2,\gamma_2} \gamma_1(B_2, E_1) \end{aligned}$$

This proof is a direct consequence of properties of composition of BIP composition operators. Thus, all the conditions imposed on contract-based verification frameworks hold. We now focus on circular reasoning.

Theorem 3 (Circular reasoning). *The framework $(\mathcal{B}, \mathcal{P}, \Gamma, \|\,.\,\|, \sqsubseteq)$ allows circular reasoning.*

Proof. Let us suppose that $B \sqsubseteq_{A,\gamma} G$ and $E \sqsubseteq_{G,\gamma} A$, that is, that (1) $Tr(\gamma(B, A)) \upharpoonright \gamma \subseteq Tr(\gamma(G, A)) \upharpoonright \gamma$ and (2) $Tr(\gamma(E, G)) \upharpoonright \gamma \subseteq Tr(\gamma(A, G)) \upharpoonright \gamma$. We have to prove that $B \sqsubseteq_{E,\gamma} G$, i.e. that $Tr(\gamma(B, E)) \upharpoonright \gamma \subseteq Tr(\gamma(G, E)) \upharpoonright \gamma$. This proof is based on the fact that in this framework all transitions are controled by exactly one component.

We examine a common prefix p of traces $Tr(\gamma(B, A)) \upharpoonright \gamma$, $Tr(\gamma(G, A)) \upharpoonright \gamma$, $Tr(\gamma(E, G)) \upharpoonright \gamma$ and $Tr(\gamma(B, E)) \upharpoonright \gamma$. We show that any possible continuation of p in $Tr(\gamma(B, E)) \upharpoonright \gamma$ is also a continuation in the other traces. A transition $\alpha = \alpha_B | \alpha_E$ that can possibly extend p in $Tr(\gamma(B, E)) \upharpoonright \gamma$, is either controled by B, or by E. In the former case, there is a corresponding transition $\alpha_B | \alpha_A$ extending p in $Tr(\gamma(B, A)) \upharpoonright \gamma$, and from (1), we know that $\alpha_B | \alpha_A$ also extends p in $Tr(\gamma(G, A)) \upharpoonright \gamma$. From (2), we get that the previous properties are true for $\alpha_A = \alpha_E$. Hence the result. The proof for the latter case is similar.

We show now how Theorem 3 and 4 of [9] are direct consequences of application of circular reasoning.

Theorem 4 (Theorem 3 of [9]).

$$\forall I_1, I_2, I_1 \sqsubseteq_{E_1,\gamma_1} S_1 \wedge I_2 \sqsubseteq_{E_2,\gamma_2} S_2 \implies \gamma_{E,2}(E, I_2) \sqsubseteq_{I_1,\gamma_1} E_1 \wedge \gamma_{E,1}(E, I_1) \sqsubseteq_{I_2,\gamma_2} E_2$$

is equivalent to

$$\gamma_{E,2}(E, S_2) \sqsubseteq_{S_1,\gamma_1} E_1 \wedge \gamma_{E,1}(E, S_1) \sqsubseteq_{S_2,\gamma_2} E_2$$

Proof. The left-to-right implication is trivial since $S_1 \sqsubseteq_{E_1,\gamma_1} S_1 \wedge S_2 \sqsubseteq_{E_2,\gamma_2} S_2$ (as for all $E, \gamma, \sqsubseteq_{E,\gamma}$ is reflexive). Now let us suppose that the following properties hold.

$$\begin{cases} \gamma_{E,2}(E, S_2) \sqsubseteq_{S_1,\gamma_1} E_1 & (1) \\ \gamma_{E,1}(E, S_1) \sqsubseteq_{S_2,\gamma_2} E_2 & (2) \\ I_1 \sqsubseteq_{E_1,\gamma_1} S_1 & (3) \\ I_2 \sqsubseteq_{E_2,\gamma_2} S_2 & (4) \end{cases}$$

We have to prove that $\gamma_{E,2}(E, I_2) \sqsubseteq_{I_1,\gamma_1} E_1 \wedge \gamma_{E,1}(E, I_1) \sqsubseteq_{I_2,\gamma_2} E_2$. By applying circular reasoning to (3) and (1), and to (4) and (2), we get the following:

$$\begin{cases} I_1 \sqsubseteq_{\gamma_{E,2}(E,S_2),\gamma_1} S_1 & (5) \\ I_2 \sqsubseteq_{\gamma_{E,1}(E,S_1),\gamma_2} S_2 & (6) \end{cases}$$

Then, by compatibility of $\sqsubseteq$ with composition, we get from (5) and (6):

$$\begin{cases} \gamma_{E,1}(E, I_1) \sqsubseteq_{S_2,\gamma_2} \gamma_{E,1}(E, S_1) & (7) \\ \gamma_{E,2}(E, I_2) \sqsubseteq_{S_1,\gamma_1} \gamma_{E,2}(E, S_2) & (8) \end{cases}$$

Finally, as $\sqsubseteq_{S_2,\gamma_2}$ and $\sqsubseteq_{S_1,\gamma_1}$ are transitive, (1) and (8) on the one hand, and (2) and (7) on the other hand, imply the result.

Theorem 5 (Circular reasoning implies independent implementability).

$$\gamma_{E,2}(E, S_2) \sqsubseteq_{S_1,\gamma_1} E_1 \wedge \gamma_{E,1}(E, S_1) \sqsubseteq_{S_2,\gamma_2} E_2$$

implies

$$\forall I_1, I_2, I_1 \sqsubseteq_{E_1,\gamma_1} S_1 \wedge I_2 \sqsubseteq_{E_2,\gamma_2} S_2 \implies \gamma_{1,2}(I_1, I_2) \sqsubseteq_{E,\gamma} \gamma_{1,2}(S_1, S_2)$$

Proof. This theorem expresses the sufficient condition for dominance provided in [13] where there are only two components with contracts (E_1, γ_1, S_1) and (E_2, γ_2, S_2), and where the global contract is $(E, \gamma, \gamma_{1,2}(S_1, S_2))$. The left-hand side in the theorem above and $\gamma_{1,2}(S_1, S_2) \sqsubseteq_{E,\gamma} \gamma_{1,2}(S_1, S_2)$ (implied by reflexivity of $\sqsubseteq_{E,\gamma}$) corresponds exactly to the precondition of our sufficient condition for dominance (see theorem 1). Hence $(E, \gamma, \gamma_{1,2}(S_1, S_2))$ dominates $\{(E_1, \gamma_1, S_1), (E_2, \gamma_2, S_2)\}$, which is exactly the right-hand side of the implication.

The proofs given in [10] depend on the formalism of I/O automata, whereas our proofs do not. They are thus longer and less easy to understand. In particular, the authors call the validity of their theorem 3 "very fortunate", while this is a natural consequence of circular reasoning. We believe that defining a general framework (as we have done in [13]) and applying it afterwards to particular frameworks (as done in this paper) helps better understanding of the key features of a given framework.

4 Application to modal I/O automata for interface and product line theories

[11] provides a unified interface theory for game models and modal transition systems. Game models specify who has *control* over transitions, while modal transition systems focus on modality, that is, which moves are allowed and which moves are required. The authors show that control information is essential to compositional reasoning and then build an interface theory based on modal automata enriched with I/O information. We adhere to this view. Yet, as in our framework control information is a first-class entity, there is no need for us to define a new set of behaviors such as modal I/O automata: we simply use modal automata, and add the information about inputs and outputs at the interactional layer. Thus, the modal interface theory developed in [11] is simply a subclass of the framework presented as an example in [13], where behaviors are modal transition systems, and composition is not restricted to connectors with at most one trigger (corresponding to the output). Separation between behavior and interaction is preserved in a natural way. In addition, we keep assumption and guarantee separate, which is a key feature for reusability of components.

5 Conclusion and future work

We have shown that using a parametrized framework in order to define an interface theory helps focusing on the key definitions, namely, (1) what representation of behaviors should be used, (2) how these behaviors should be composed, (3) what refinement in a context means. These definitions should be decided carefully in order to ensure the properties expected from the framework, e.g. circular reasoning.

We have illustrated this claim by discussing two papers where circular reasoning is extensively used without naming it, thus making the design process more difficult to understand.

Use of our parametrized framework reveals decisions that are often implicit and that we think should be made explicit. In particular, if circular reasoning is not sound in a particular framework, there are two possibilities: either not use it, or find another definition of refinement in a context such that in the modified framework circular reasoning is sound (and unfortunately not complete, see [12]). Such a choice should be made carefully because it has a strong influence on the subsequent verification methods to be used.

We think that an interesting application of this work would be to fit synchronous approaches for contract-based verification into our framework. We believe that using a unified framework for comparing numerous interface theories should allow to detect more key properties of these theories, and thus help design of new ones.

References

1. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time systems in bip. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06), Invited talk, September 11-15, 2006, Pune, pp 3-12*, 2006.
2. Albert Benveniste, Benoît Caillaud, and Roberto Passerone. A generic model of contracts for embedded systems. *CoRR*, abs/0706.1456, 2007.
3. Simon Bliudze and Joseph Sifakis. The algebra of connectors: structuring interaction in bip. In *EMSOFT*, pages 11–20, 2007.
4. Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In *CONCUR*, pages 508–522, 2008.
5. Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proc. of the 9th Annual Symposium on Foundations of Software Engineering, FSE'01*. ACM Press, 2001.
6. Willem-Paul de Roever, Frank de Boer, Ulrich Hannemann, Jozef Hooman, Yassine Lakhnech, Mannes Poel, and Job Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge, UK, November 2001.
7. G. Goessler and J. Sifakis. Composition for component-based modeling. *Science of Computer Programming*, pages 161–183, March 2005.
8. Kim Guldstrand Larsen. Modal specifications. In *Automatic Verification Methods for Finite State Systems*, pages 232–246, 1989.
9. Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Interface input/output automata. In *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 82–97, 2006.
10. Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Interface input/output automata. Technical report, BRICS, 2006.
11. Kim Guldstrand Larsen, Ulrik Nyman, and Andrzej Wasowski. Modal i/o automata for interface and product line theories. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4421 of *Lecture Notes in Computer Science*, pages 64–79, 2007.
12. Patrick Maier. Compositional circular assume-guarantee rules cannot be sound and complete. In *FoSSaCS*, pages 343–357, 2003.
13. Sophie Quinton and Susanne Graf. Contract-based verification of hierarchical systems of components. In *IEEE International Conference on Software Engineering and Formal Methods (SEFM08)*, 2008.

An Aspect-Oriented Behavioral Interface Specification Langauge

Takuo Watanabe*
Department of Computer Science
Tokyo Institute of Technology

Kiyoshi Yamada
Research Center for Information Security
National Institute of Advanced Industrial Science and Technology

Abstract

We have designed and developed a behavioral interface speci cation language Moxa. that provides a modularization mechanism for contracts based on assertions. The mechanism, called assertion aspect, can capture the cross-cutting properties among assertions. In this paper, we briefly explain the notion of assertion aspects and the design of Moxa. By comparing the speci cation to its JML counterpart, we show that the use of assertion aspects clari es the large, complex speci cation and greatly simpli es each assertion in the speci cation.

1 Introduction

Design by Contract (DbC) is a software development method that utilizes assertions in a principled manner. In DbC, the "contract" between a class and its clients is a set of conditions (pre-/postconditions of the methods and a class invariant) typically represented as assertions embedded in the source code. The contract provides the detailed interface speci cation of the class.

DbC is especially bene cial for developing reliable software systems. The authors have experience in applying DbC to the actual development of a working application in which reliability is the prime factor to be considered. The application — AnZenMail client — is a secure and reliable e-mail client implemented in Java. It is a part of the AnZenMail system [8], an experimental testbed for cutting-edge security enhancement technologies. The AnZenMail system has been developed by a group of researchers involved in the research project "*Research on Implementation Schemes for Secure Software*" supported by Japanese Ministry of Education, Culture, Sports, Science and Technology.

The primary purpose of applying DbC was to ensure the code quality of the AnZenMail client. To ensure the code quality of the AnZenMail client, we rst wrote a formal speci cation of its important component, called the Maildir Provider, that should handle received e-mails and mail folders in a reliable way. We used the Java Modeling Language (JML) [6] to describe its speci cation with DbC-style

*takuo@acm.org

assertions. With this speci cation, we checked the component thoroughly using the JML tools and then we could nd bugs in the code (including Sun's JavaMail components) and the assertions. This process, which was actually performed incrementally and repeatedly, enabled us to gradually obtain solid code and the rm speci cation of the component. The nal speci cation consists of approximately 3,500 lines of assertions.

While we were carrying out the above process, we often observed the following problem: changes made to an assertion in a class caused the propagation of changes in the assertions within other classes. In principle, DbC assertions in a class are independent from ones in other classes. But in real life, while we were working with some large, seemingly unrelated classes, we often encountered the above phenomenon. This can be a serious obstacle for developing, maintaining or extending a large-scale software with DbC. We have observed that there are properties that span over the assertions in several program modules (classes or methods). The problem comes from the fact that the coverage of such properties does not t the inherent structure made from the program modules. In other words, they *crosscut* the modules.

To overcome the problem, we introduced a new modularization mechanism for assertions that aims to separate the crosscutting properties. The mechanism is based on *assertion aspect*, a new notion in aspect-oriented technology. So far, we have designed a new behavioral interface speci cation language *Moxa*, an extension of JML, that provides the mechanism[10].

The rest of the paper is organized as follows. In the next section, we introduce the notion of assertion aspect and our behavioral speci cation language Moxa. Then, in Section 3, we compare Moxa to JML by using the same example. Section 4 mentions the related work. Section 5 o ers a discussion of the results.

2 Assertion Aspects in Moxa

2.1 Crosscutting Properties

As a part of the AnZenMail[8] client (mentioned in the previous section), we developed the *Maildir Folder Service Provider* (*Maildir Provider* for short), a JavaMail [9] component. It speci es the structure for directories of incoming e-mail messages and can provide reliable hierarchical mailboxes by using sophisticated algorithms for handling message les.

We used the Java Modeling Language (JML) [6] to describe the speci cation of the Maildir Provider. In the speci cation we wrote, however, assertion expressions become complicated and bulky. The size of the nal Java code of the Maildir Provider and its JML speci cation (without the code) are about 2,500 and 3,500 lines respectively. This makes it di cult to develop the code and the speci cation incrementally with keeping the consistency of assertions and code.

The source of the problem is the mismatch of modularization structures between the assertions and the Java code. In JML, we write assertions as annotations associated to classes and methods. This forces that assertions are grouped into the modules enforced by the language — in this case, classes and methods. But this is not always appropriate for the modularization of assertions.

```
1   public spec S {
2     public behavior
3       requires Pre1;
4       ensures Post1;
5     Ta C1.m1(T1 x1, ...);
6     Tb C2.m2(T2 x2, ...);
7
8     public behavior
9       requires Pre2;
10      ensures Post2;
11    Tc C3.m3(T3 x3, ...);
12    Td C4.m4(T4 x4, ...);
13  }
```

Figure 1: An Assertion Aspect in Moxa

2.2 Aspects in AspectJ

Aspect-oriented Programing (AOP) [5] is a programming technique for modularizing concerns that cross-cut the modules in programs. Some kind of code fragments related to concerns such as logging, synchronization, exception handling or performance optimization, are mingled within functional modules. In other words, they cross-cut the modules. AspectJ [4] is an extension of Java that provides a mechanism for modularizing such tangled code. The key notions of the mechanism are *pointcut* and *advice*. A pointcut is a set of *join points* that are particular locations on the control ow of the program. An advice is a pair of pointcut and a code fragment executed at the location selected by the pointcut. An aspect consists of a set of advice.

2.3 Assertion Aspects in Moxa

The notion of aspect in Moxa is di erent from the one in AspectJ. The di erence is that an aspect in Moxa is applied to speci cations (logical expressions written as annotations), while an aspect in AspectJ is applied to code. We call aspects in Moxa *assertion aspects* to avoid confusion with aspects in AspectJ.

Figure 1 shows that how an assertion aspect is de ned. In this de nition, `S` is the name of this assertion aspect, `C1` $\cdots$ `C4` are class names, `m1` $\cdots$ `m4` are method names, `x1` $\cdots$ `x4` are identi ers (arguments) and `Ta` $\cdots$ `Td`, `T1` $\cdots$ `T4` are type descriptors. `Pre1` and `Pre2` (`Post1` and `Post2`) are pre-conditions (post-conditions) respectively.

An assertion aspect is a collection of advice (as in AspectJ). Figure 1 has two advice: lines 2–6 and lines 8–12. The advice is a pair of a pointcut and an assertion condition. The pointcut is a set of join points that are locations on the control ow of a program. The location on the control ow where we want to test the pre- or post-condition of the constructors or the methods, *pre-* and *post-condition location* respectively and we call them *assertion locations*. Because the assertion in Moxa is based on DbC, a join point is normally identical to the assertion location. A descriptions of pointcuts (e.g., lines 5–6) consists of a set of method signatures and positional keywords `requires` (or `ensures`). The rst advice (lines 2–6) in Figure 1 describes two pointcuts at once that show the pre-condition location of method `m1` and `m2`, and the post-condition location of these methods.

```
public spec FolderState {
  public behavior
    requires chkState_connected(...)
    ensures chkState_eq(...)
  public int Folder.getMessageCount()
  throws MessagingException;

  public behavior
    requires chkState_open(...)
    ensures chkState_eq(...)
  public Message Folder.getMessage(int msgnums)
  throws MessagingException;

  public behavior
    requires chkState_closed(...)
    ensures chkState_open(...)
  void Folder.open(*) throws MessagingException;

  public behavior
    requires chkState_open(...)
    ensures chkState_closed(...)
  void Folder.close(*) throws MessagingException;

  ...
}
```

Figure 2: An Assertion Aspect Specifying State Transition of Folders (abridged)

A join point in Moxa corresponds to a location in the ordinary assertion declaration technique where the assertion declaration is inserted. In the ordinary assertion declaration technique, when we want to describe the same assertion in two or more assertion locations, we have to describe assertions for each of those assertions locations. On the other hand, in Moxa, we can describe the condition of these assertions only once by an advice whose pointcut selects these assertion locations.

2.4 Example

Figure 2 shows an assertion aspects that speci es the state transition of the class `Folder`. This assertion aspect captures and modularizes a concern on the states of folders. In this example, the logical expression in each pre-/post-condition consists of the invocation of a method such as `chkState_open`. These methods are de ned in actual classes (thus they are implementation dependent) and provide actual state information. This makes the assertion aspect `FolderState` implementation independent.

3 Evaluation

In this section, we compare Moxa to JML by using the same example. The target of the speci cations is a part of the *Maildir Folder Service Provider* (*Maildir Provider*

Table 1: Comparison of the Two Speci cations

	JML		Moxa	
	`Service`	`Store`	`Service`	`Store`
# of Modules	1	1	3	5
# of Assertions	42	53	13	18
# of Lines	190	149	152	286
# of Lines / Module	190	149	51	57

for short) that is a part of the AnZenMail client mentioned in Section 1. The Maildir Provider is a JavaMail [9] component that manages *maildir* style mailboxes on le systems. We compared the speci cations of two classes `Store` and `Service` that are de ned in the abstract layer of JavaMail . The items of comparison are the number of modules (the number of classes in JML and the number of assertion aspects in Moxa), the number of assertions (the number of pre- and post conditions in JML and the number of advice in Moxa), and the number of lines (comments included). The result of comparison is shown in Table 1, and its characteristics are described below.

Number of Modules: In the case of JML, the number of modules for each class is 1 because a modularization unit of JML must be matched to the class or interfaces. In the case of Moxa, the number of modules are 3 and 5 for the class `Service` and `Store`, respectively. This is because, each crosscutting condition of assertion can be split into di erent assertion aspects.

Number of Assertions: In the case of JML, the number of assertions are 42 and 53 for the class `Service` and `Store`, respectively. In the case of Moxa, the number of assertions are 13 and 18 for the class `Service` and `Store`, respectively, and each number is smaller than the case of JML. This is because crosscutting conditions over the assertions includes the same logical expressions, and they can be organized into an advice in Moxa.

Number of Lines in Assertions: The number of lines in assertion descriptions in JML are 190 and 149 for the class `Service` and `Store` respectively. On the other hand, the total number of lines in assertion aspects of the Moxa speci cation are 152 and 286, for the class `Service` and `Store` respectively. Thus, we can see that the average number of lines in an assertion aspect is much smaller than the average number of lines in the JML speci cation. This comes from the fact that the same logical expression of assertions for some join points are merged into one advice in Moxa using pointcuts.

This result shows that using Moxa, the size of each module in a speci cation will be reduced. We can also expect that this can clarify large and complex speci cations by modularizing crosscutting properties that span over the program modules.

Locality of Changes: Table 2 shows the e ect of a simple change in the code. Here, we replace the method `boolean Service.isConnected()` to `boolean Service.notConnected()`. The table summarizes the e ect of this change on the speci cations: the number of the modules (classes in JML and assertion aspects in Moxa) we should x and the number of lines possibly to be a ected. In the Moxa speci cation of the class `Service` (`Store`), we should only change 6 (4) modules. Please note that we don't need to examine the rest of the modules. The number

Table 2: Number of Changes in the Speci cations

	JML		Moxa	
	`Service`	`Store`	`Service`	`Store`
# of Changes	42	53	6	4
# of Lines in Changes	190	149	54	40

of assertions and the number of lines to be changed dramatically decreases, because of aspect-orientation. This result shows that Moxa provides higher locality in speci cation.

4 Related Work

Injecting assertion validation code into application modules is a typical application of AOP. There have already been several proposals on describing assertions using AspectJ [7, 2, 3]. They point out the problems of embedding assertions in the program code and propose ways to describe assertions separately from program code. Especially, Lippert and Lopes [7] investigate that global properties on exception detection and handling can be systematically represented using AspectJ.

Though writing validation code in AspectJ is one possible way to modularize assertions, it is generally complex and error prone task. Moreover, this style of assertion description is specialized to runtime validation. This means that using assertions with other analysis/veri cation tools is di cult.

Since Moxa has a dedicated syntax, speci cations written in this language can be used not only for runtime validation, but also with other tools. Currently we are implementing Moxa processor as a translator to JML. Thus, it is possible to use existing JML tools.

Contract4J [1] is another tool that supports DbC in Java. This tool provides annotation based syntax for assertions and uses AspectJ for injecting validation code.

Pipa [12] is an extension of JML whose target language is AspectJ. With this language, we can describe assertions for the AspectJ constructs such as advice or introduction. However, as in JML, assertions in Pipa are modularized within target language (AspectJ) modules; i.e., classes or aspects. This means that Pipa does not provide modularization of crosscutting properties. Extending Moxa to support AOP languages is future work.

5 Discussion

5.1 Modularization of Assertions

The simple assertion description technique for object-oriented programming language based on DbC such as JML has no mechanisms to control the mapping between assertions and methods. So, specifying pre- and post-conditions are permitted at most once a method, and they must be modularized by the unit of classes. On the other hand, Moxa enables us to describe assertions independently of the program structure considering assertion assignment location consists of a class, a method, and pre- or post-condition locations as pointcut and assertion description

as advice. In the technique, for example, the following style of assertion declarations are permitted.

- Specifying assertions to a class from one or more assertion aspects.
- Specifying one or more assertions to an assertion location (logical expression of these assertions are associated with logical product).
- Specifying assertions to one or more classes from one assertion aspect.

Using Moxa, we can split the behavior of object or object group into several independent sides, and we can describe each side of behavior into separated assertion aspects. This feature holds the scale and complexity of assertion aspects small. Moreover, the viewpoint of each assertion aspect becomes narrowed to some simple side. Hence, expressing and understanding the meaning of an assertion aspect becomes easy. Also, the maintainability and quality of assertion aspects and corresponding programs are improved.

5.2 From Incremental Re nement to Model-Driven Development

In Moxa, we can describe JML annotations along with assertion aspects, because Moxa is an extension of JML. Therefore, Moxa enables us not only to modularize assertions as assertion aspects independent of the programs structure, but also to specify assertions as annotations embedded into the program. Such a feature is favorable for the incremental development. Concretely, we can specify assertions using in-place annotations for the program code at the early stage of development or modi ed rapidly. Then, the code becoming stable and crosscutting properties are unveiled, we can extract assertion aspects from annotations. This process can be used for incremental re nement of existing code.

For example, suppose that we can extract an assertion aspect (say A_1) from a speci cation of an existing system. And suppose that A_1 captures the state transition of modules in the system (as in Figure 2) If A_1 can be re ned to A_2 that represents a more reliable state model[1], the we can re-apply A_2 to the original code and validate it to re ne the code itself. This process can gradually improves the reliability of existing code.

Moreover, assertion aspects may represent other models. A sort of model-driven development (as in [11]) might be possible by using appropriate tools that generate a code skeleton from an assertion aspect. We have been extending Moxa to include the support of protocol-oriented aspect description. The extension provides convenient way to describe, test and verify speci cations that are described based on method invocation sequence.

6 Concluding Remarks

This paper presented the notion of assertion aspects and a new behavioral interface speci cation language Moxa that provides a modularization mechanism for assertions. The mechanism enables us to separate crosscutting properties spanning over multiple assertions. It can clarify a large, complex speci cation and also can greatly

[1] Here, the term model denotes the notion in MDD.

simplify the assertions in the speci cation by eliminating common logical subexpressions. Assertion aspect broadens the scope of AOP by providing the separation of speci cation concerns, instead of code concerns.

References

[1] Aspect Research Associates, *Contract4J*, http://www.contract4j.org.

[2] Diotalevi, F., *Contract enforcement with AOP: Apply design by contract to Java software development with AspectJ*, IBM developerWorks (2004), http://www-106.ibm.com/developerworks/library/j-ceaop.

[3] Ishio, T., T. Kamiya, S. Kusumoto and K. Inoue, *Assertion with aspect*, in: *International Workshop on Software Engineering Properties for Aspect Technologies (SPLAT2004)*, 2004.

[4] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. G. Griswold, *An overview of AspectJ*, ECOOP 2001, LNCS **2072**, 2001, pp. 327–355.

[5] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin, *Aspect-oriented programming*, in: *ECOOP '97: Object-Oriented Programming*, LNCS **1241**, 1997, pp. 220–242.

[6] Leavens, G. T., A. L. Baker and C. Ruby, *JML: A notation for detailed design*, in: H. Kilov, B. Rumpe and W. Harvey, editors, *Behavioral Specifications for Businesses and Systems*, Kluwer, 1999 pp. 175–188.

[7] Lippert, M. and C. V. Lopes, *A study on exception detection and handling using aspect-oriented programming*, ICSE 2000, ACM, 2000, pp. 418–427.

[8] Shibayama, E., S. Hagihara, N. Kobayashi, S. Nishizaki, K. Taura and T. Watanabe, *AnZenMail: A secure and certified e-mail system*, in: *Software Security: Theories and Systems*, LNCS **2609** (2003), pp. 201–216, 2003.

[9] Sun Microsystems, *JavaMail API*, http://java.sun.com/products/javamail.

[10] Yamada, K. and T. Watanabe, *An Aspect-Oriented Approach to Modular Behavioral Specifications*, ENTCS, 163 (1), pp. 45–56, Sep., 2006.

[11] Zhang, J., J. Gray and Y. Lin, *A model-driven approach to enforce crosscutting assertion checking*, in: *Workshop on Modeling and Analysis of Concerns in Software (MACS 2005)*, 2005.

[12] Zhao, J. and M. Rinard, *Pipa: A behavioral interface specification language for AspectJ*, in: *Fundamental Approach to Software Engineering (FASE 2003)*, LNCS **2621** (2003), pp. 150–165.

Treaty - A Modular Component Contract Language

Jens Dietrich, Graham Jenson
School of Engineering and Advanced Technology (SEAT)
Massey University, Palmerston North, New Zealand
Email: j.b.dietrich@massey.ac.nz,grahamjenson@maori.geek.nz

Abstract

In recent years, dynamic component-based systems such as OSGi and its derivatives have become very successful. This has created new challenges for verification. Assemblies are created and modified dynamically at runtime, but many existing techniques such as unit testing are designed for buildtime verification. Runtime verification is usually restricted to type checks. We propose a simple component contract language that is powerful enough to represent different types of complex contracts between collaborating components, including contracts with respect to component semantics and quality of service attributes, and contracts that refer to resources other than programing language artefacts. These contracts can then be used for runtime verification of assemblies. Contracts are based on a pluggable contract vocabulary. We present a proof of concept implementation of the contract language proposed for the OSGi/Eclipse component model.

1 Introduction

Component-based systems have become very popular in the last decade. While initially used mostly in desktop application, component-based software engineering is now used in many different areas including server-side and ubiquitous computing. This has created a number of new challenges for component models with respect to component lifecycle and resource management. Traditionally, component models focus on one particular aspect to describe the relationship between collaborating components - interface compatibility. This relationship is defined by a contract that is usually expressed by means of programming language artefacts like Java interfaces, or by using a dedicated interface definition language (IDL). However, modern component models have to address use cases where other types of contracts are involved. For instance, server applications often require a high level of reliability, and applications running on mobile devices have special requirements with respect to the (hardware) resources components can use. If components are dynamically discovered, it might not be enough to know that these components provide the right interface, they must also have the expected behaviour. Beugnards at al. [BJPW99] have investigated types of component contracts and have classified contracts into four layers:

1. Basic syntactic contracts expressing interface compatibility.
2. Behavioural contracts specifying component semantics.
3. Synchronisation contracts describing dependencies between components.
4. Quality of service contracts describing requirements with respect to response times, quality of results etc.

Beugnards at al. have also discussed several technologies that could be used to express contracts for the various layers. This includes the use of IDLs for layer 1 and design by contract [Mey92] for layer 2. Other types of contracts not covered by this classification include aspects related to security, trust and licensing. For instance, an organisation might want to prevent the use of components with contagious licenses, or configurations where components with incompatible licenses are linked together.

In some modern component frameworks even basic layer 1 contracts can be rather complex. A good example is the successful component model used by Eclipse [Ecl]. Based on OSGi [OSG] bundles, Eclipse plugins use extension points and extensions to define required and provided resources. Often, these resources are Java types - plugins define extension points using Java interfaces, and require other plugins providing extensions to these extension points to supply classes implementing the respective interfaces. However, in general these contracts are highly polymorphic. An example for this is the `org.eclipse.help.toc` extension point. In order to extend it, applications have to provide help resources and a table of content XML file instantiating a given document type definition. Moreover, many extension points use complex logical expressions. An example for this is `org.eclipse.ui.actionSets`. Here, the value of the attribute `class` must be a name of a class that implements an interface. Which interface this is depends on the value of another attribute (`style`).

In this paper, we introduce Treaty, a component contract language designed to address these issues. The paper is organised as follows: In section 2, we summarise the Treaty contract language, we use an Eclipse-based example application for this purpose. This application contains polymorphic and disjunctive contracts, and uses unit test cases for layer 2 and layer 4 contracts. We then discuss contract instantiation and verification. In section 4 we show how contract vocabularies can be organised in a modular manner. In section 5 we explore the use of unit test cases in contracts in more detail. We then discuss the architectural aspects of Treaty, focusing on the relationship between contracts and the underlying component model. A discussion of related work and open questions concludes our contribution.

The Treaty framework and the example used throughout this paper are both accessible on Google code[1], the code is licensed under the Apache open source license.

[1] http://code.google.com/p/treaty/, the Eclipse update site URL is http://treaty.googlecode.com/svn/trunk/treaty-eclipse-updatesite/site.xml. The Treaty plugin requires JDK 1.6 or better.

2 Formalising Contracts

Components collaborate in different ways. When designing component-based systems in an object-oriented language, the most common way of collaboration is that one component provides an abstract type, while another component provides (an instance of) an implementation class of this abstract type. The use of abstract types decouples the collaborating components. As mentioned in the introduction, modern component-based systems like Eclipse use also different types of contracts. For instance, components have to supply XML documents instantiating document type definitions (DTDs) or XML Schemas. In general, we can consider the artefacts provided and consumed by components as resources identified by uniform resource identifiers (URIs). These resources are typed, examples for types are instantiable Java classes, Java interfaces, IDL interfaces, XML instances, XML Schemas, XSL files, DTDs, property files, and CSV files. Relationships associating resources are defined for certain resource types only, for instance Java classes implement Java interfaces, XML documents instantiate DTDs, or style sheet transformations applying to instances of a certain XML Schema.

In [DHG07] it has been proposed to use the semantic web standards RDF[KC04], OWL[MvH04] and SWRL[HPSB+04] to model component contracts in a platform-independent manner. While this has some obvious advantages, including the existence of a formal semantics for SWRL, the resulting rules are too complex and do not support a compact representation of contracts. Furthermore, these contracts have restricted expressiveness. In particular, complex constraints using exclusive disjunctions cannot be represented. For this reason, we have developed a custom XML vocabulary that supports the compact definition of component contracts. This vocabulary is part of Treaty, the contract framework we propose. Contracts define the relationship between two parties: consumer and supplier. Treaty as a framework abstracts from the concrete nature of these entities. For the example used here we use the proof of concept implementation of Treaty for the Eclipse component model. Here, the consumer and supplier roles are mapped to extension points and extensions, respectively.

Figure 1 shows such a contract [2]. The respective example is implemented as a set of Eclipse plugins. In this contract, the relationship between a component that prints dates (clock view) and a component that provides a date formatting service (date to string) is defined. The contract is attached to the Eclipse component that has the extension point as an XML file in the component meta-data folder (`META-INF`). The name of this file is defined by the following naming convention: the name of the extension point followed by the extension ‘‘`.contract`’’. This mechanism is non-invasive - contracts can be added to plugins without modifying existing plugin resources. Treaty does not modify the Eclipse plugin registry either - it is only queried through public interfaces and if there are no contracts found for an extension point it is interpreted as empty contract.

A Treaty contract has three parts:

1. In the consumer section (lines 3-19), the resources of the extension point are defined. The resources defined are constants identified by name and type. The types are defined in an (external) ontology and represented by

[2] The package names are abbreviated

URIs. This information can be used by the component to load resources if needed, for instance by using the component class loader.

2. In the supplier section (lines 20-27), the resources of the extension are defined. This is where a component provides resources to be consumed by a consumer. These resources are also typed. Resources are now variables, the `ref` element is used to define a variable that can be used to query for the resource once a concrete extension is known. This reflects the support for dynamic component models that use late binding. Details of this mechanism are discussed further below.

3. In the constraints part (lines 28-45), the relationships between resources of both sides are specified. The schema supports the use of standard logical connectives such as AND, OR and XOR to define complex conditions. In addition to relationships, value properties and existence conditions are supported as well.

In the example shown in figure 1, the clock component that has the extension point provides the following resources (package names for classes omitted):

1. The interface `DateFormatter` (id `‘‘Interface’’`) that describes the interface of the date formatter service.

2. The `dateformat.xsd` (id `‘‘DateFormatDef’’`) schema that describes the interface of an alternative service by means of an XML schema. Instances of this schema define date formatting string templates.

3. The class `DateFormatterFunctionalTests` (id `‘‘FunctionalTests’’`) defines some JUnit functional test cases. The test cases check whether the strings produced by a date formatter contain at least the day, the month (as number or using the English name of the month) and the last two digits of the year. They define the minimal information content of strings rendering dates.

4. The class `DateFormatterPerformanceTests` (id `‘‘QoSTests’’`) defines JUnit quality of service tests. It checks whether a date formatter needs less than 10ms to render a date.

The extending component must provide one of two resources: a Java class or an XML document. The contract conditions state that a valid extension must either provide an XML instance that is valid with respect to the schema, or an instantiable class that implements the interface and passes additional functional and performance tests.

Conditions in contracts can be either atomic or complex. To build complex conditions, the usual logical connectives with their standard semantics can be used. Three types of atomic conditions are supported: relationships between resources, resource properties, and conditions that a resource must exist. Relationships and properties are equivalent to object and data properties in RDF. The `mustExist` constraint is weaker - this merely asserts that the respective resource must exist and must be of the declared type.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <contract>
3   <consumer>
4     <resource id="Interface">
5       <type>http://www.treaty.org/java#AbstractType</type>
6       <name>clock.DateFormatter</name>
7     </resource>
8     <resource id="QoSTests">
9       <type>http://www.treaty.org/junit#TestCase</type>
10      <name>clock.DateFormatterPerformanceTests</name>
11    </resource>
12    <resource id="FunctionalTests">
13      <type>http://www.treaty.org/junit#TestCase</type>
14      <name>clock.DateFormatterFunctionalTests</name>
15    </resource>
16    <resource id="DateFormatDef">
17      <type>http://www.treaty.org/xml#XMLSchema</type>
18      <name>/dateformat.xsd</name></resource>
19  </consumer>
20  <supplier>
21    <resource id="Formatter">
22      <type>http://www.treaty.org/java#InstantiableClass</type>
23      <ref>/serviceprovider/@class</ref></resource>
24    <resource id="FormatString">
25      <type>http://www.treaty.org/xml#XMLInstance</type>
26      <ref>/serviceprovider/@formatdef</ref></resource>
27  </supplier>
28  <constraints>
29    <xor>
30      <and>
31        <relationship
32          resource1="Formatter" resource2="Interface"
33          type="http://www.treaty.org/java#implements"/>
34        <relationship
35          resource1="Formatter" resource2="FunctionalTests"
36          type="http://www.treaty.org/junit#verifies"/>
37        <relationship
38          resource1="Formatter" resource2="QoSTests"
39          type="http://www.treaty.org/junit#verifies"/>
40      </and>
41      <relationship
42        resource1="FormatString" resource2="DateFormatDef"
43        type="http://www.treaty.org/xml#instantiates"/>
44    </xor>
45  </constraints>
46 </contract>
```

Figure 1: XML Contract Example

condition	semantics
property	comparison of a property of a resource with a literal using a comparison operator
relationship	establishes whether a relationship exists between resources
mustExist	true if the referenced resource exists

Table 1: Basic contract condition types

3 Contract Lifecycle and Verification

The sample contract is still abstract since it references resources (the resources of the supplier) that are not yet known at the time the contract is written. The supplier is only known later at runtime when late binding occurs. Only then the contract can be instantiated. Contract instantiation is the creation of a deep copy of the contract, and the instantiation of all resource proxies in this copy. A resource proxy is a resource that has a `ref` attribute but no `name` attribute. The `ref` attribute is a reference to the components meta-data. The Treaty framework contains an interface `ResourceManager` that is used to resolve those proxies. The details of resolving are component-model specific. In the Eclipse-based implementation of Treaty, the `ref` values are XPath expressions and the `ResourceManager` uses them to query the plugin meta-data (`plugin.xml`). In an implementation of Treaty for pure OSGi the attribute values could just be simple strings representing keys of properties defined in the bundle manifests.

Once an instantiated contract exists, verification can be performed. This is the checking of conditions according to their semantics. When complex conditions are present, this is usually done using a top-down strategy. This is a simple process: once all resources are instantiated, contracts are essentially statements of classical propositional logic. The question is how the basic conditions are checked. This requires the resources to be loaded. For instance, to check properties of a resource of the type `AbstractType`, the respective class must be loaded so that it can be analysed using the Java reflection framework. This is done with a `ResourceLoader`. Again, on the framework level this is an interface that must be implemented when adapting Treaty for a particular component model. In case of Eclipse, the loader uses the OSGi bundle classloader to load resources.

4 Contract Vocabularies

Contracts reference types and properties. Both can be formally defined in a formal ontology language, but this alone does not define their semantics [Usc01]. For instance, the semantics of the (Java) `implements` predicate (figure 1, line 33) is the set of pairs of concrete Java classes C and Java interfaces I such that C implements I. In other terms, the semantics can be defined by a function that takes two resources C and I and can compute a boolean indicating whether $(C, I) \in \mathit{implements}$ is the case or not. This particular function is easy to provide: if C and I can be loaded and are available as instances of `java.lang.Class`, the method `isAssignableFrom` can be used to check this condition. In a similar manner, a validating XML parser can be used to check the `instantiates` property associating XML instances and XML schemas.

A possible solution to this problem is to define a fixed type system that contains a set of commonly used resource types, and implements some classes that represent the semantics of their properties and relationships. However, there might be very project-specific types and relationships to be used in contracts. Consider a scenario where a company has a product with a reporting extension point. This offers customers the option to plug-in their own reporting templates with customised layouts and data aggregation. The resource type to be provided by these components could be `VelocityTemplate`[Vel]. Or even better, a project-specific `MyReportTemplate` type that represents velocity templates that use only a fixed set of variables which the host component can bind. Then the component itself would make contributions to the contract vocabulary in order to enable verification. There is a clear business use case for this: it safeguards the company against faulty third party plugins which would result in customers blaming the company for the malfunctioning of their software.

Therefore, the vocabulary should be kept open and extensible. This can be achieved by using the component model itself to build modular contract vocabularies. Each vocabulary component must provide the following:

1. A list of defined types (URIs) contributed by the component.
2. A list of defined properties (URIs) contributed by the component.
3. A list of defined relationships (URIs) contributed by the component.
4. A method to load a resource given a reference and a resource type. For instance, this method is used to load resources of the type Java class defined by an attribute in `plugin.xml` as Java classes using the plugin's class loader.
5. A method that can be used to check the properties and relationships contributed by the component.

In the Treaty implementation for Eclipse, this functionality is defined through the extension point `net.java.treaty.eclipse.vocabulary`. To extend this extension point, plugins must implement an interface that has the methods to load resources and check conditions, and have to provide an OWL resource that defines the vocabulary extensions. Treaty merges the ontology contributions into a central merged ontology. This ontology contains all contributed types, properties and relationships, plus annotation indicating which component contributed the respective artefacts.

The reporting template example shows the benefits of using formal ontologies. For instance, assume that the reporting template type `MyReportTemplate` subclasses `VelocityTemplate`, and that the contract requires only the existence of a reporting template. Because of the semantics of `rdfs:subClassOf` the verifier could than first check whether the resource is of the type `VelocityTemplate` by using the Velocity parser. If this fails, the resource cannot be an instance of `MyReportTemplate` either. That is, the formal semantics of OWL can be used to optimise verification.

For this reason, in the proof of concept implementation all components making vocabulary contributions have access to a central singleton `Vocabulary` that maintains the virtual merged ontology. This allows them to use ontology reasoning when checking contributed properties and relationships. The ontology

can be accessed as unparsed stream or as java object representing the parsed ontology.

5 Unit Testing at Runtime

The example contract (figure 1) uses the `verifies` property to express minimum requirements with respect to functionality and performance for classes implementing the `DateFormatter` interface. This relationship is based on JUnit, that is, the test resources are JUnit 4 test cases, and their semantics is defined by means of a JUnit test runner. JUnit test cases are defined in the same component that defines the date formatter interface. These tests check whether date formatter implementations can convert dates in less than 10ms, and whether the generated strings contain at least tokens representing date, month and year.

Unit testing is particularly useful here as it stands in the tradition of design by contract - describing the semantics of methods through a description of the state changing effects of the methods expressed by pre- and post conditions. The main weakness of unit testing when compared to other verification methods is that verification is based on selected specimen objects. Tests are not sufficient to prove or ensure correctness, they can only be used to approximate it. The main advantage of unit tests is that they are widely acceptance by programmers. Also, it is easy to assess the degree of approximation (coverage metrics), and there are well-established development processes to improve test cases when it is necessary to improve the approximation.

Unfortunately, JUnit has been built for design and build time verification. As a consequence of this it is assumed that the classes to be tested are known when the test cases are written and can be directly referenced by test cases. On the other hand, our approach supports late binding at composition time, that is, test cases can only reference abstract types and the actual objects have to be injected if the respective classes become available at runtime. Therefore, JUnit needs to be modified to fit into Treaty. More precisely, support for dependency injection mechanism must be added to JUnit. This is achieved by designing test cases that have constructors with parameters that can be used to inject the tested objects before the test case life cycle starts, and a special test runner that can instantiate test cases using this constructor. Such a test runner is part of the Treaty component that makes the JUnit vocabulary contributions.

6 The Bigger Picture - Adding Contracts to Component Models

Treaty is implemented in Java and provides support for contract definition and verification for the Java-based Eclipse component model. However, Treaty is largely independent of the underlying component model and could also be used to describe contracts in other component models even if they are not Java-based. Treaty itself can be seen as a combination of three separate subsystems:

1. The Contract Definition Language (CDL), a formal language used to define contracts in a platform-independent manner. In this paper we have

proposed to use XML (constraint by the treaty.xsd schema) for this purpose.

2. The Contract Execution Environment (CEE), a system that reads contracts defined in the CDL and can instantiate and verify the contracts against components of a host component model. The CEE proposed here is implemented in Java and consists of two parts - an abstract contract framework and an implementation of the abstract concepts in the framework for the OSGi/Eclipse component model.

3. The Contract Vocabulary (CV), an ontology that defines the types and properties that are used in contracts.

The CEE must reference the CM to instantiate resource references using the reflective features of the CM (such as access to meta-data). It also uses the CM to load resources needed to verify constraints. Finally, the CEE provides the semantics for the (data and object) properties used in the vocabulary. The CEE has access to the merged ontology and can use it for ontology reasoning.

Our Eclipse-based implementation adds two more relationships: both the CV and the CEE take advantage of the CM to define both the vocabulary and the parts of the CEE providing the semantics for the vocabulary in a modular fashion.

7 Discussion

We have presented Treaty, a component framework that supports the easy definition of complex and polymorphic contracts. Our main contribution is the contract language, and the modular design of the contract vocabulary. We believe that using such a language adds value to environments that use late binding, such as ubiquitous or mobile computing applications where new components are discovered and integrated at runtime. The types of requirements that need to be expressed in environments like this are somehow unpredictable. We therefore think that using any fixed contract language is not appropriate. Instead, what is needed in an extensible contract language based on a platform-independent description of resource types and their relationships. This allows components to plugin vocabulary extensions that can then be used by verification tools.

Treaty is still rather simple, and simplicity was one of the major design goals when designing Treaty. One reason for this is of course the fact that much of the work is delegated to the vocabulary contributions. However, in many cases it is rather easy to write these contributions, and the level of reuse for vocabulary elements would be much higher than the level of reuse of the actual application components. The main advantage is that such an open framework supports a consistent representation of different contract types by using a common meta-model (OWL). To the best of our knowledge, no existing (academic or industrial) component models or architectural description language achieves this.

In the prototype we have presented, verification is used as a central service that checks the integrity of the entire system. It might be more useful in many circumstances to check only contracts between certain components, for instance in response to lifecycle events such as component activation. An interesting issue is whether contracts should be attached to components consuming resources (as

we have done this), to components providing resources or should be detached from either ("contracts as entities in the middle", as proposed in [Szy00]. On the framework level, Treaty does support contracts on both sides and in the middle, and the aggregation of multiple contracts. The proof of concept implementation based on Eclipse however only support contracts on the consumer side at the moment.

References

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[DHG07] Jens Dietrich, John Hosking, and Jonathan Giles. A Formal Contract Language for Plugin-based Software Engineering. In *Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 175–184, Washington, DC, 2007. IEEE Computer Society.

[Ecl] Eclipse. http://www.eclipse.org.

[HPSB+04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. W3C member submission, W3C, May 2004. http://www.w3.org/Submission/SWRL/.

[KC04] Graham Klyne and Jeremy J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C recommendation, W3C, February 2004. http://www.w3.org/TR/2004/REC-owl-features-20040210/.

[OSG] The OSGi Alliance. http://www.osgi.org.

[Szy00] Clemens Szyperski. Components and contracts. Dr Dobbs, May 2000. http://www.ddj.com/architect/184414613.

[Usc01] Michael Uschold. Where is the Semantics in the Semantic Web? In *Workshop on Ontologies in Agent Systems*, Montreal Canada, May 2001.

[Vel] The Apache Velocity Project. http://velocity.apache.org/.